Entwicklung, Anwendung und Analyse eines neuronalen Netzes zur Erkennung von Streumustern in Röntgenuntersuchungen

Simon Jung

Bachelorthesis
zur Erlangung des akademischen Grades

Bachelor of Science
Physik

Bergische Universität Wuppertal

Erstgutachter:
Prof. Dr. Uwe Pietrzyk

Zweitgutachter:
Dr. Markus Axer

Abgabe:
Wuppertal, den 25. März 2019

Abstract

Die vorgestellte Thesis entwickelt ein neuronales Netz zur Korrektur von Streumustern in Röntgenprojektionen.
Die Grundlagen des maschinellen Lernens mit neuronalen Netzen werden erörtert. Mit der Simulationssoftware GATE werden Röntgenprojektionen simuliert. Durch eine Filterung anhand der maximalen Weglänge zwischen Quelle und Detektor wird die Streustrahlung in den Projektionen abgeschätzt. Die Datenpaare aus simulierten Projektionen und bestimmter Streustrahlung werden als Trainingsdaten für ein zweischichtiges neuronales Netz verwendet. Das neuronale Netz sowie Funktionen zur Datenverarbeitung werden entwickelt und implementiert.
Das Training und die Anwendung des neuronalen Netzes wird anhand simulierter Röntgenprojektionen analysiert und bewertet. Die Analyse und Bewertung finden optisch anhand von Details in den Projektionen statt.
Das Netz kann die Streustrahlung in geometrisch unähnlichen Projektionen reduzieren. Abhängig der verwendeten Trainingsdaten wird das statistische Bildrauschen reduziert und die Auflösung von Strukturen der Größe weniger Pixel verschlechtert.
Die Anwendung von neuronalen Netzen zur Beschreibung physikalischer Zusammenhänge wird diskutiert.

Inhaltsverzeichnis

1 Einleitung

Die Röntgenbildgebung ist in der medizinischen Diagnostik und in der zerstörungsfreien Materialprüfung ein etabliertes Verfahren, um Strukturen innerhalb des menschlichen Körpers oder eines Objektes aufzulösen. Die sogenannte Streustrahlung bezeichnet Röntgenstrahlung, die aufgrund eines Prozesses im durchstrahlten Objekt ihre Richtungsinformation verliert. In der Messung ist diese Strahlung von der ungestreuten Strahlung zunächst nicht zu unterscheiden. Hierdurch wird die eigentliche Strukturabbildung gestört.
Streustrahlung entsteht zwangsläufig, da sie Teil des bildgebenden Prozesses ist. Streustrahlung verringert die Auflösung in der entstehenden Projektion und erzeugt Bildfehler. Durch eine Reduktion der Streustrahlung lassen sich Röntgenaufnahmen verbessern. Dies führt zu einer höheren Detailauflösung [1].

Zur Reduktion bzw. zur Erkennung von Streustrahlung wird in dieser Arbeit ein neuronales Netz entwickelt, welches mit simulierten Daten trainiert wird. Zur Simulation wird die Software GATE verwendet, welche mit Monte Carlo-Methoden Röntgenprojektionen erzeugen kann. In diesen Simulationen wird der Streustrahlungsanteil mithilfe der Simulationsinformationen bestimmt. Somit entstehen Datenpaare von Röntgenprojektionen und dem enthaltenen Anteil Streustrahlung.

Es wird ein neuronales Netz erstellt und mit den Datenpaaren trainiert, um den Streuanteil in Röntgenprojektionen bestimmen oder abschätzen zu können.
Ein neuronales Netz ist eine Methode des maschinellen Lernens. Das neuronale Netz dient dazu, auf Basis bekannter Zuordnungen von Datensätzen die Merkmale zu erkennen, welche möglichst allgemeingültige Regeln für die Zuordnung erlauben.
Es werden der Aufbau und die Methoden eines solchen neuronalen Netzes erörtert und implementiert. Das erstellte Programm wird auf die simulierten Röntgenprojektionen angewendet.
Die Ergebnisse werden analysiert und mit aktuellen Entwicklungen in diesem Bereich verglichen. Es wird diskutiert, wie die Aussagefähigkeit und die Anwendbarkeit eines solchen neuronalen Netzes bewertet werden kann.

1.1 Motivation

Die Röntgenbildgebung wird in der Medizin und Industrie vielfältig eingesetzt. Im Besonderen bietet die Anwendung in der Computertomographie vielfältige Möglichkeiten zur medizinischen Diagnose, der Werkstoffprüfung, Metrologie und Defektanalyse.
Bei der sogenannten Computertomographie werden unter mehreren Winkeln Röntgenprojektionen eines Objekts oder eines Patienten angefertigt. Aus diesen Aufnahmen können Schichtbilder und dreidimensionale Modelle des untersuchten Objekts erstellt werden [2].
Die Anforderungen an Röntgenaufnahmen sind vielfältig. Die Qualität einer Röntgenaufnahme

wird häufig anhand der geometrischen Auflösung von Objekten bestimmt. Die Möglichkeit, Materialen voneinander zu unterscheiden, ist eine wichtige Eigenschaft. Streustrahlung kann dazu führen, dass feine Strukturen nicht erkannt werden können und Materialgrenzen nicht lokalisierbar sind [1].

Streustrahlung hat siginifikanten Einfluss auf die Güte von medizinischen Diagnosen und Objektuntersuchungen. Streustrahlung entsteht zwangsläufig in der Röntgenbildgebung als objektabhäniger Prozess. Aufgrund der Objektabhängigkeit lässt sich bei bekannter Objektausdehnung und -zusammensetzung die Streustrahlung durch Monte Carlo-Methoden sehr gut abschätzen [3]. Da sowohl die geometrischen Informationen, als auch die Materialinformationen durch eine Röntgenprojektion aufgenommen werden, wurde die Vermutung aufgestellt, dass ein neuronales Netz erstellt werden kann, welches diese Informationen direkt zur Streustrahlerkennung nutzbar macht. Ein neuronales Netz würde insofern ein vielfältig anwendbares Werkzeug darstellen, um Röntgenaufnahmen und somit auch die Diagnosen und Untersuchungen mit geringem Rechenaufwand zu verbessern [4].

2 Einführung in die Röntgenbildgebung

1895 wurde durch Wilhelm Conrad Röntgen die Röntgenbildgebung entdeckt und seitdem auf verschiedenste Arten und Weisen weiterentwickelt [5]. Die Röntgenbildgebung hat weitreichende Anwendungsbereiche, von der medizinischen Diagnostik über die Strukturanalyse bis hin zu Materialprüfung und Metrologie. Die inneren Strukturen von Objekten werden durch unterschiedliche Absorption elektromagnetischer Wellen aufgelöst.

2.1 Entstehung von Röntgenstrahlen

Als sogenannte Röntgenstrahlen werden elektromagnetische Wellen mit Wellenlängen zwischen ungefähr 10 nm (Energie $\sim$ 100 eV) und 1 pm (Energie $\sim$ 1 MeV) bezeichnet [6]. Röntgenstrahlen entstehen meist durch beschleunigte Elektronen. Geben Elektronen kinetische Energie in Form elektromagnetischer Wellen bzw. Photonen ab, spricht man von Röntgenstrahlung.
Hierbei sind zwei Effekte zu berücksichtigen. Die sogenannte Bremsstrahlung entsteht, wenn Elektronen durch elektromagnetische Wechselwirkung mit anderen Teilchen abgelenkt werden und dabei einen Teil ihrer kinetischen Energie in Form von Röntgenphotonen abgeben [7]. Die maximale Energie dieser Bremsstrahlung entspricht der kinetischen Energie des Elektrons, also einer kompletten Abbremsung.
Entfernt ein beschleunigtes Elektron durch einen Stoß ein gebundenes Elektron eines Atoms aus dessen diskretem Energiezustand, wird das frei werdende Energieniveau in der Regel durch ein gebundenes Elektron höherer Energie aufgefüllt. Die Energiedifferenz der Energieniveaus wird in Form eines Photons abgestrahlt. Da die Elektronen nur quantisierte Energieniveaus des Atoms einnehmen, werden nur bestimmte Energien abgestrahlt. Die entstehende Energieverteilung wird als diskretes Röntgenspektrum bezeichnet [6].
Röntgenstrahlung entsteht also, wenn bewegte Elektronen auf Materie treffen. Das gesamte abgestrahlte Spektrum setzt sich aus diskreten Energiewerten und der kontinuierlichen Bremsstrahlung zusammen.
Zur praktischen Erzeugung von Röntgenstrahlung werden Elektronen beschleunigt, welche auf ein Target treffen. Als Target dienen typischerweise Materialien wie Kupfer oder Wolfram, an deren Atomen die Bremsstrahlung sowie die charakteristische Strahlung entstehen. Die Elektronen werden in der Regel direkt durch ein elektrisches Feld zwischen einer heißen Kathode und dem Target als Anode erzeugt. Die angelegte Spannung gibt die maximale Energie der Röntgenstrahlung vor [1].

2.2 Bildgebung mit Röntgenstrahlen

Die Bildgebung mit Röntgenstrahlung basiert auf materialabhängiger Absorption bzw. Streuung der Röntgenphotonen. Wird Röntgenstrahlung kegelförmig aus einer näherungsweise punktförmigen Quelle auf ein Objekt gestrahlt, entsteht hinter dem Objekt ein zu dieser Absorption und Streuung passendes Intensitätsmuster. Diese Intensitätsverteilung wird durch Fluoreszenzschirme oder Detektoren sichtbar gemacht. Hier entstehen passend zur Objektzusammensetzung helle

und dunkle Bereiche. Um diese Bereiche dem Objekt konkret zuordnen zu können, ist es wichtig, dass sich die den Schirm erreichenden Photonen auf geraden Trajektorien bewegen. Werden die Photonen durch Streuung abgelenkt und erreichen dennoch den Detektor, verfälschen sie die Objektinformation [1].
Die Streuung und Absorption sind auf verschiedene Effekte zurückzuführen. Zu nennen sind hier die Streuung von Photonen (Compton-Streuung und Rayleigh-Streuung) und die absorbierenden Prozesse (Paarbildung und der Photoeffekt).

2.2.1 Streustrahlung als Abbildungsfehler

Je mehr Abbildungsfehler in einer Röntgenprojektion vorhanden sind, desto weniger Aussagen lassen sich über das untersuchte Objekt treffen. Um ein Detail oder eine Objektcharakteristik zu erkennen, muss es von Streustrahlung unterscheidbar sein. Die Streustrahlung überlagert das reine Absorptionsbild des Objektes und führt so zu falschen Werten im Abbild [1].
Lässt sich der Einfluss von Streustrahlung auf eine Projektion nicht minimieren, so müssen meist mehr Aufnahmen angefertigt werden oder eine höhere Energie verwendet werden. Dies führt zu einer Dosis- und Gesamt-Messzeit-Erhöhung.
In der Röntgenbildgebung existiert stets die Maxime die Aufnahmezeit zu minimieren. Röntgenbildgebung wird hauptsächlich in der Medizin an lebenden Menschen angewendet und kann durch Ionisierung zur Schädigung von Erbgutinformation führen. In der industriellen Anwendung können rauschreduzierte Aufnahmen die Objektuntersuchung beschleunigen und ermöglichen dadurch beispielsweise eine in-line-Anwendung zur Qualitätskontrolle in Prozessketten der Produktherstellung. Die Streustrahlreduktion führt zu einer höheren Auflösung, was beispielsweise bei Detailuntersuchungen nach dem Versagen eines Bauteils von hoher Wichtigkeit ist.
Vor diesem Hintergrund wird in der vorliegenden Arbeit die Tauglichkeit von Machine Learning Algorithmen zur Erkennenung und dem Filtern von Streustrahlung untersucht (siehe Abschnitt 8). Diese Untersuchung findet auf Basis eines Multi-Layer-Perceptron-Ansatzes statt (siehe Abschnitt 7). Ein Ausblick auf die Einbindung weitergehender physikalischer Ansätze wird formuliert (siehe Abschnitt 9).

2.2.2 Rayleigh-Streuung

Rayleigh-Streuung bezeichnet die Streuung eines Photons an einem Atom. Aufgrund der hohen Masse des Atoms wird das Photon praktisch ohne Energieverlust gestreut. Betrachtet man das Photon als elektromagnetische Welle, so regt es ein gebundenes Elektron an. Dieses strahlt infolgedessen in derselben Wellenlänge ab, mit der es angeregt wurde. Entscheidend ist, dass die elektromagnetischen Wellen wie bei einem Hertzschen Dipol abgestrahlt werden und somit die Richtungsinformation des Photons unbrauchbar wird.
Der Streuquerschnitt für Rayleigh-Streuung hängt direkt von ω^4, der vierten Potenz der Frequenz des Photons, ab. Insofern sinkt die Wahrscheinlichkeit für Rayleigh-Streuung für höher energetische Photonen und ist bei Röntgenstrahlung kein dominierender Effekt [7].

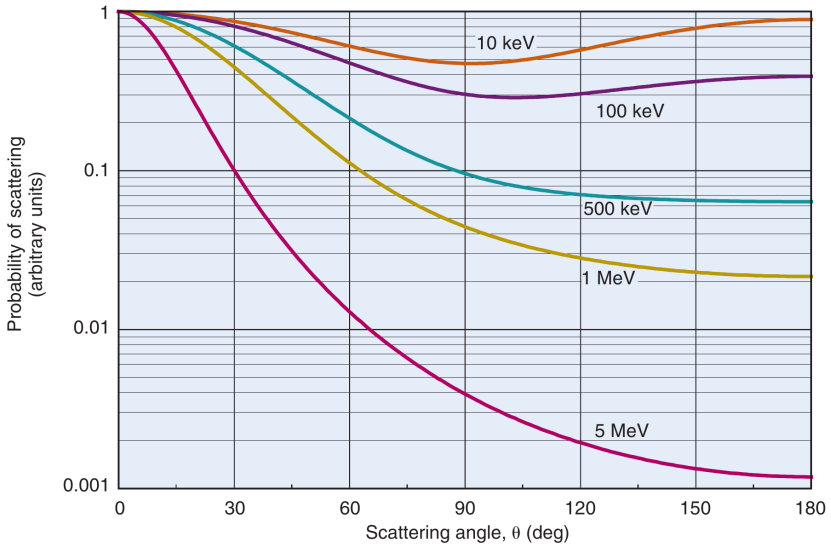


Abbildung 2.1 Wahrscheinlichkeit der Comptonstreuung unter einem bestimmten Winkel für verschiedene Energien.[7, S. 76]

2.2.3 Compton-Effekt

Der Compton-Effekt beschreibt den elastischen Stoß zwischen einem Photon und einem Teilchen. Dabei wird ein nicht-diskreter Teil der Energie des Photons auf das Teilchen übertragen. Die übertragene Energie hängt aufgrund der Impulserhaltung von dem Winkel ab, unter dem die Streuung stattfindet. Durch die Energieabgabe erfährt das Photon eine Wellenlängenänderung, welche lediglich von dem Streuwinkel und nicht von der Energie des Photons abhängig ist. Die Wellenlängenänderung ist ein Anteil der Compton-Wellenlänge des Teilchens, welcher bei Rückstreuung ($\sim 180°$) maximal und bei keiner oder nahezu keiner Ablenkung ($\sim 0°$) minimal ist. Die Compton-Wellenlänge des Teilchens ist die Wellenlänge, welche der Ruheenergie des Teilchens entspricht [7].
Mit Zunahme der Energie sinkt die Wahrscheinlichkeit für eine Rückstreuung und die Wahrscheinlichkeiten für geringe Streuwinkeln nehmen zu [7]. Dieser Zusammenhang ist gut in Abbildung 2.1 zu erkennen.

2.2.4 Photoeffekt

Bei dem sogenannten Photoeffekt löst ein Photon ein gebundenes Elektron aus seiner Bindung und wird dabei selber absorbiert. Die Energie des Photons muss somit mindestens der Bindungsenergie des Elektrons entsprechen und wird vollständig an das Elektron übergeben. Der frei gewordene Energiezustand wird in der Regel durch ein höherenergetisches Elektron des gleichen Atoms eingeommen. Die dabei frei werdende Energie wird wieder in Form eines Photons abgestrahlt [7].

2.2.5 Paarbildung

Die Paarbildung beschreibt die Entstehung eines Teilchen-Antiteilchen-Paares. Es kann zur Paarbildung kommen, wenn die Energie des Photons mindestens die Ruheenergien der beiden entstehenden Teilchen umfasst. Hier ist nur die Bildung von Elektron-Positron-Paaren aus Photonen

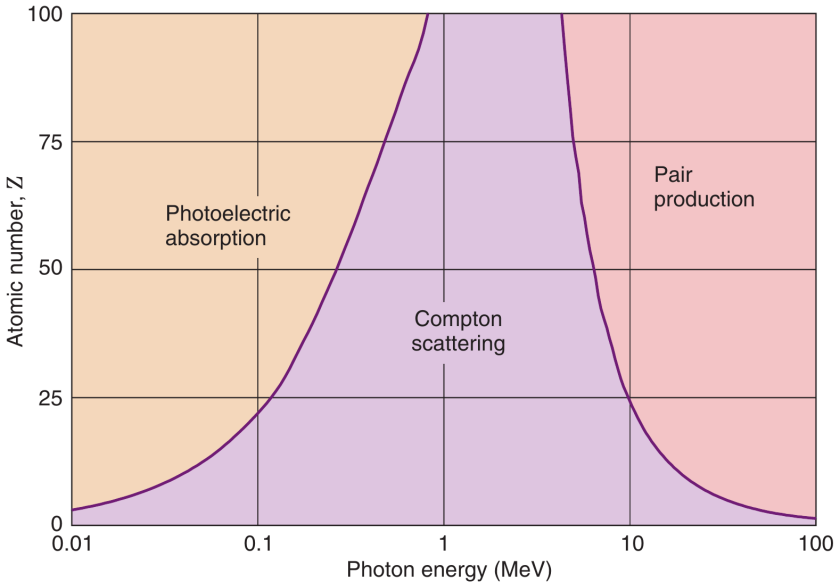


Abbildung 2.2 Auftragung der dominierenden Effekte für Photonen in Materie, aufgeschlüsselt nach Energie und Ordnungszahl.[7, S. 81]

relevant. Da die benötigte Energie hier ~ 1.022 MeV enstspricht, tritt die Parrbildung erst bei Linearbeschleunigern auf, die entsprechend hochenergetische Photonen liefern [7]. Abbildung 2.2 zeigt, in welchem Energiebereich und für welche Ordnungszahlen die beschriebenen Effekte jeweils dominieren. Es ist zu erkennen, dass im Energiebereich der Röntgenstrahlen zunächst die Comptonstreuung und für höhere Ordnungszahlen der Photoeffekt dominant sind.

3 Einführung zur Simulationssoftware GATE

GATE (Geant4 Application for Tomographic Emission) ist eine Software der INTERNATIONAL OPENGATE COLLABORATION. Die Software dient zur Simulation verschiedener bildgebender Verfahren, wie Computer-Tomographie (CT), Einzelphotonen-Emissions-Computer-Tomographie (SPECT) und Positronen-Emissions-Tomographie (PET) [8, 9, 10]. Die Software wird Open Source entwickelt [11]. Zur Dokumentation der einzelnen Funktionen existiert ein Wiki [12].
GATE basiert auf der GEANT4 Software, welche am CERN entwickelt wird, um das Verhalten von Teilchen in Materie zu simulieren [13, 9]. Als Grundlage dienen Monte Carlo-Simulationen und Datenbanken der physikalischen Prozesse. GATE wird durch sogenannte Makros bedient. Diese dienen dazu einzelne Funktionanlitäten einzubinden und miteinander zu kombinieren. Zur Datenausgabe dient hauptsächlich die Software ROOT, welche ebenfalls am CERN entwickelt wird [14].
In den für diese Bachelorarbeit durchgeführten Simulationen wird GATE in der Verion 8.1.01, GEANT4 in der Version 10.03.03 und ROOT in der Version 6.10.04 verwendet.

3.1 Grundlegende Struktur

Die Steuerung durch Makros hat den Hintergrund, dass GATE modular und einfach anwendbar sein soll. So können Konzepte verschiedener tomographischer Methoden übergreifend angewendet werden. Der User hat keinen direkten Zugriff auf die physikalischen Mechanismen, die zur Simulation genutzt werden (vgl. Abbildung 3.1). Es können lediglich verschiedene Datenbanken zur Verwendung ausgewählt werden [15].

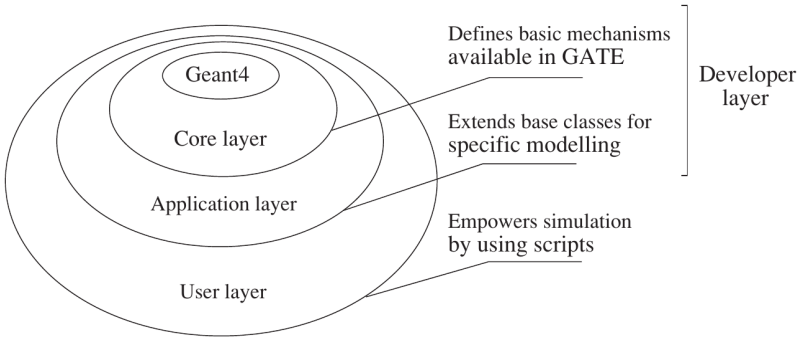


Abbildung 3.1 Eine schematische Darstellung des Aufbaus von GATE. [9, S. 4547]

3.2 Aufbau einer Simulation

Ein GATE-Skript basiert auf einigen fundamentalen Elementen. Hier sind zu nennen: die Geometrie des Detektors, des Phantoms und der Strahlungsquelle(n), die Definition der Strahlungsquelle(n) (Spektrum, Intensität), die Einstellung der zu berücksichtigenden physikalischen Effekte sowie die Definition der genauen Auslesemechanismen des Detektors. Zusätzlich werden noch die Objektbewegungen, die Messzeiten, die Datenausgabe und weitere formale Dinge festgelegt.

4 Einführung in das maschinelle Lernen mit neuronalen Netzen

Das maschinelle Lernen dient dazu mithilfe von Computern und Algorithmen, verallgemeinerte Regeln aus Zusammenhängen in Daten zu erhalten. Diese Trainingsdaten umfassen möglichst viele Objekte, die einer Aussage/Klassifizierung zugeordnet sind. Diese Objekte bestehen aus bestimmten Merkmalen, die für jedes Objekt erfasst sind. Ein Algorithmus versucht einen möglichst allgemeingültigen Zusammenhang zwischen den einzelnen Merkmalen und der jeweiligen Klassifizierung herzustellen. Hierbei fließen von Seiten des Algorithmus lediglich statistische Eigenschaften der Daten ein. Um eine logische, physikalische oder sonstige kausale Aussage miteinfließen zu lassen müssen explizite Randbedingungen definiert oder die Merkmale entsprechend gewählt werden.

4.1 Historische Entwicklung

Die ersten Ansätze maschinellen Lernens basieren auf der Funktionsweise einer Nervenzelle. Bereits 1943 haben W. Pitts und W. McCulloch ein logisches Modell der Nervenzelle aufgestellt, welches 1957 von F. Rosenblatt als Netz zum ersten lernfähigen Algorithmus nachvollzogen wurde [16, 17].

4.1.1 Zugrundeliegendes Modell der Nervenzelle

Eine Nervenzelle besteht unter anderem aus einem Körper (Soma) und einem Axon. Die Soma ist über die sogenannten Dendriden mit anderen Zellen an deren Synapsen verbunden und das Axon gibt Signale der Nervenzelle an andere Zellen weiter. Die Dendriden übertragen Signale lediglich von anderen Zellen auf die Nervenzelle und das Axon überträgt Signale lediglich von der Nervenzelle auf andere Zellen. Die eingehenden Signale kumulieren sich in kurzen Zeitintervallen. Überschreitet die entstehende kumulierte Summe ein konkretes Niveau, so gibt die Nervenzelle ein Ausgangssignal ab. Die Entstehung eines Ausgangssignals wird als Aktivierung bezeichnet und das entstehende Signal kann nur einen Wert annehmen [16, 18, 19].
W. Pitts und W. McCulloch haben 1943 ihre Annahmen über Netze von Nervenzellen in fünf Punkten zusammengefasst. Es wird die Aktivierung eines Neurons als binärer Prozess, welcher durch eine konkrete Anzahl aktiver Synapsen ausgelöst wird, beschrieben. Der Aufbau des Netzes ändert sich nicht mit der Zeit. Zeitliche Verzögerungen enstehen nicht in der Nervenzelle sondern nur in den Verbindungen, den Synapsen. Es existieren sogenannte *inhibitory synapses*, die, wenn sie aktiv sind, die Aktivierung der Nervenzelle verhindern [16]. Die Nervenzelle entspricht somit einem logischen Gatter, welches eine logische 1 ausgibt, wenn genügend Eingangssynapsen aktiviert sind und andernfalls (bzw. im Falle einer aktivien *inhibitory synapse*) kein Signal sendet (logische 0) [19].

4.1.2 Entwicklung des Perzeptrons

F. Rosenblatt entwickelte 1957 das Modell des Perzeptrons [17]. Danach wird ein System entwickelt, welches aus drei Ebenen, einer Eingabe-, einer Zuordnungs- und einer Ausgabeebene besteht. Die Eingabeebene umfasst eine binäre Dateneingabe. Jeder Punkt dieser Ebene ist mit einem oder mehreren Punkten der Zuordnungsebene verbunden. Jeder Punkt der Zuordnungsebene erzeugt ein Signal, wenn die Summe, der mit dem Punkt verknüpften Eingabesignale eine Schwelle θ übersteigt. Die Verbindungen zwischen Eingabe- und Zuordnungsschicht können negativ oder positiv sein, dass heißt ein Eingangssignal kann negativ oder positiv in die Summe eines Punktes der Zuordnungsebene einfließen. Jedem Punkt in der Zuordnungsebene ist ein variabler Parameter v der Ausgabe zugeordnet. Dieser Parameter entspricht bei Rosenblatt direkt einer physikalischen Größe wie einem Widerstand oder einer Kapazität [17].
Die Ausgabeebene erzeugt eines oder mehrere Ausgabesignale. In einem Punkt der Ausgabeebene sind ein oder mehrere Punkte der Zuordnungsebene summiert oder gemittelt verknüpft. Der entstehende Wert aktiviert die entsprechende Ausgabeeinheit ebenfalls durch Überschreitung eines Schwellenwertes θ_r. Es bestehen weitere Verknüpfungen, die gleichzeitige Ausgaben verschiedener Werte über eine Ausgabeeinheit verhindern und die Ausgabe, hervorgerufen durch den höheren Wert in der Zuordnungsebene, bevorzugen.
Der Lernprozess findet dadurch statt, dass, solange Punkte in der Zuordnungsebene eine Ausgabe hervorrufen, die Variablen v aller an der Ausgabe beteiligten Zuordnungseinheiten erhöht werden. Wird eine Ausgabe zu einer konkreten Eingabe erzwungen, also wird die gewünschte Ausgabeeinheit extern aktiviert, erhöhen sich die Variablen v aller Zuordnungseinheiten, die in der gewählten Eingabe-Konstellation zur extern aktivierten Ausgabe führen. Dadurch wird in Zukunft bei gleicher Eingabe die erwartete Ausgabe durch das System erzeugt. Problematisch ist hierbei, dass verschiedene Eingabe-Ausgabe-Daten über dieselben Zuordnungseinheiten laufen und gegebenenfalls widersprüchliche Werte für die Variablen v hervorrufen. Das führt dazu, dass das System nicht beliebig viele verschiedene Zuordnungen erlernen kann.
Rosenblatt zeigt weiter, dass sich das Modell mit elektrischen Schaltungen realisieren lässt [17]. In Abschnitt 4.2.2 wird weiter auf das Perzeptron-Modell eingegangen.

4.1.3 Weitere Entwicklungen

Die Struktur des Perzeptrons bildet ein sogenanntes neuronales Netz. Hierbei stehen in der Eingabeebene Merkmale, die zu einer Klassifizierung in der Ausgabeebene führen sollen. Die Neuronen entsprechen den Elementen der Zuordnungsebene und sind mit verschiedenen Elementen der Ein- und Ausgabeebene vernetzt. Sie verfügen über Gewichte, ähnlich den Variablen, die den Einfluss der einzelnen Merkmale auf die Klassifizierung festlegen.
Auf Basis des Perzeptrons wurde 1960 das sogenannte Adaline (Adaptive linear Neuron) entwickelt. Dieses unterschiedet sich vom Perzeptron, indem die Optimierung der Gewichte des Netzes nicht auf Basis der korrekten oder falschen Klassifizierung (Heaviside-/Schwellenwert-Funktion) geschieht, sondern anhand einer linearen Aktivierungsfunktion, bevor die eigentliche Klassifizierung (weiterhin über eine Heaviside-Funktion) geschieht (siehe hierzu die mathematische Darstellung der Funktion in Gleichung (4.2) und den entsprechenden Plot in Abbildung 4.2). Ein wichtiger Punkt ist hierbei die Einführung des Gradientenabstiegsverfahrens (siehe Abschnitt 4.2.2) [20]. Heutzutage werden meist logistische Funktionen für die Anpassung der

Gewichte verwendet (vgl. Abschnitt 4.2.2).
Eine weitere populäre Erweiterung neuronaler Netze ist das Hinzufügen weiterer Schichten. Solche Netze werden als Deep Neural Networks bezeichnet. Vorteil ist, dass sie komplexere Strukturen abbilden können, welche auf tiefergehenden Konzepten beruhen. Durch mehrere Schichten können die einzelnene Eingabemerkmale mehrfach mit unterschiedlichen Kombinationen und Gewichtungen miteinander verknüpft werden.
Eine Weiterentwicklung sind die Convolutional Neural Networks (CNN) [19, 489ff]. Die Anwendung liegt hauptsächlich in der Objektklassifizierung in Bildern, theoretisch ist allerdings eine Anwendung auf jedes mehrdimensionale Signal möglich. CNNs fügen einem neuronalen Netz zusätzliche sogenannte Pooling- und Faltungs-Schichten hinzu. In den Pooling-Schichten wird die Information mehrerer benachbarter Pixel anhand einer Funktion zusammengefasst und in den Faltungs-Schichten wird das Bild mit einer Matrix gefaltet. Beide Operationen können konkrete Eigenschaften (bspw. Kanten im Bild oder aus einem Muster fallende Pixel) hervorheben und gleichzeitig den Umfang der zu verarbeitenden Matrizen verringen.

4.2 Grundlegende Elemente neuronaler Netze

Maschinelles Lernen durch neuronale Netze lässt sich auf verschiedene Arten und Weisen realisieren. Jedes neuronale Netz besitzt sogenannte Hyperparameter, die großen Einfluss auf die Anwendbarkeit des Netzes haben (siehe Abschnitt 4.2.6). Diese Parameter haben direkten Einfluss auf den Aufbau und die Funktion eines Lernalgorithmus. Die Analyse der Anwendbarkeit eines Algorithmus erfordert somit meist auch eine Untersuchung der optimalen Hyperparameter.

4.2.1 Unterscheidung verschiedener Lernarten

Es werden meist drei Lernarten des maschinellen Lernens unterschieden, das überwachte, das unüberwachte und das verstärkende Lernen [19, S. 26].
Das überwachte Lernen ist charakterisiert durch einen direkten Abgleich des vorhergesagten Ergebnisses mit dem erwarteten Ergebnis. Hier können zum Training nur Daten mit bekannter Klassifizierung verwendet werden. Ein überwacht trainiertes Netz dient dazu, den Trainingsdaten ähnliche Datensätze zu klassifizieren.
Das unüberwachte Lernen dagegen arbeitet mit Daten ohne bekannte Klassifizierung und kann somit auch nach dem Durchlaufen des Netzes keinen Abgleich mit einer Erwartung durchführen. Derartiges Lernen kann dennoch wichtige Erkenntnisse durch Separation verschiedener Untermengen (Clustering) und der Erkennung von Strukturen liefern.
Das verstärkende Lernen trennt die Bewertung der Zuordnungen von der direkten Klassifizierung. Beim verstärkenden Lernen existiert eine Belohnungsfunktion, welche den Zustand eines ganzen Systems beurteilen kann und nicht nur einzelne Klassifizierungen berücksichtigt. Diese Belohnungsfunktion liefert dem System Anhaltspunkte, um die Leistung zu optimieren.[19, S. 26–31]
In dieser Arbeit werden Methoden des überwachten Lernens verwendet. Es werden Datensätze erstellt, die die Eingabewerte und die angestrebten Ausgabewerte enthalten. Diese Werte werden nach Durchlaufen des Netzes miteinander verglichen und zum Verbessern des Netzes verwendet.

4.2.2 Lineare Klassifizierer

Linear trennbare Datenmengen können durch lineare Klassifizierer unterschieden werden. Viele Probleme lassen sich bereits durch eine lineare Trennung lösen. Es ist auch die Kombination mehrerer linearer Klassifizierer möglich, um mehr als zwei Klassen zu unterscheiden. Hier wird meistens ein One-vs-All Ansatz angewendet. In diesem Fall wird jede mögliche Klassifizierung einzeln gegen alle übrigen Klassifizierungen gestellt. Insofern muss für jede Klassifizierung der Klassifizierer neu trainiert werden, da stets nur eine binäre Entscheidung getroffen werden kann [19, S. 51].

Das Perzeptron

Die künstlichen Neuronen bilden das Modell der Nervenzelle von W. Pitts und W. McCulloch nach (siehe Abschnitt 4.1.1). Es wird eine binäre Klassifizierung erreicht, also die Unterscheidung zweier Klassen (meist -1 und +1).

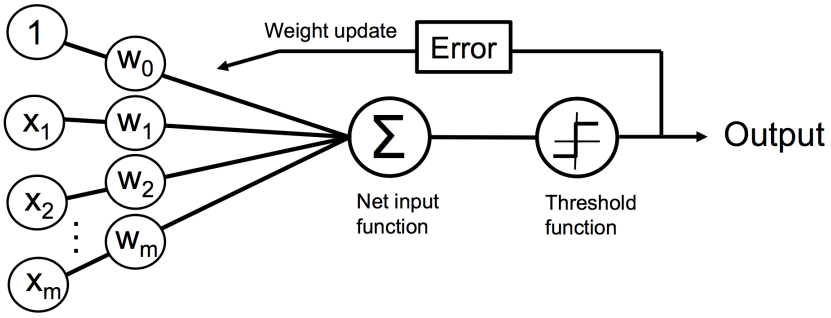


Abbildung 4.1 Schematische Darstellung des Perzeptrons. Eingabedaten x_j fließen gewichtet (w_j) in eine Nettoeingabe ein, welche mit einem Schwellenwert verglichen wird. Der Vergleich mit der gewünschten Ausgabe führt zu einer Anpassung der Gewichte. [19, S. 46]

Eingabewerte $\mathbf{x}$ werden mit Gewichten $\mathbf{w}$ versehen, um die sogenannte Nettoeingabe z zu ergeben. Es gilt

$$\mathbf{w} = \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_m \end{pmatrix}, \qquad \mathbf{x} = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_m \end{pmatrix}, \qquad z = \mathbf{w}^T\mathbf{x} \quad . \tag{4.1}$$

Es ist Konvention, dass lediglich die Elemente mit Index $1 \dots m$ den Eingabewerten und Gewichten entsprechen. Die Elemente w_0 und x_0 dienen der Definition eines Schwellenwertes. Mit $w_0 = -\theta$ und $x_0 = 1$ kann die binäre Klassifizierung als Vergleich mit dem Schwellenwert θ wie folgt über eine Aktivierungsfunktion ϕ (Heaviside-Funktion, siehe Abbildung 4.2) stattfinden

$$\phi(z) = \begin{cases} 1, & z \geq 0 \\ -1, & sonst \end{cases}, \qquad \text{wobei} \quad w_0 = -\theta, \quad x_0 = 1. \tag{4.2}$$

Jedes Objekt i besitzt einen Satz Eingabewerte (Merkmale) $\mathbf{x}^{(i)}$. Die Initialisierung der Gewichte $\mathbf{w}$ geschieht mit kleinen Zufallszahlen. Es wird mithilfe der Aktivierungsfunktion der Ausgabewert $\hat{y}$ für jedes Objekt $\mathbf{x}^{(i)}$ berechnet. Daraufhin werden die Gewichte durch Vergleich mit dem

erwarteten Wert y mit einer Lernrate η aktualisiert

$$\mathbf{w} := \mathbf{w} + \Delta\mathbf{w} \qquad \text{, wobei} \quad \Delta\mathbf{w} = \eta \left(y^{(i)} - \hat{y}^{(i)} \right) \mathbf{x}^{(i)} \quad . \tag{4.3}$$

Das Perzeptron konvergiert nur, wenn die Daten linear trennbar sind [19, S. 42–49].

Adaline - Adaptive lineare Neuronen

Die bereits 1960 entwickelte Adaline-Regel unterscheidet sich von der Perzeptron-Lernregel durch eine lineare Aktivierungsfunktion (im Gegensatz zur Heaviside-Funktion beim Perzeptron-Modell) (siehe auch 4.1.3 und Abbildung 4.2). Hier gilt $\phi(z) = z = \mathbf{w}^T\mathbf{x}$. Die Anpassung der Gewichte wird durch eine Minimierung der quadratischen Differenz der erwarteten Werte $y^{(i)}$ von der Aktivierungsfunktion realisiert. Zur Beschreibung dieses Zusammenhangs wird die sogenannte Straffunktion

$$J(\mathbf{w}) = \frac{1}{2} \sum_i \left(y^{(i)} - \phi \left(z^{(i)} \right) \right)^2 \tag{4.4}$$

eingeführt. Der Faktor $\frac{1}{2}$ ist reine Konvention zur Vereinfachung der Minimierung. Da es sich hier um eine konvexe quadratische Funktion handelt, lässt sich die Optimierung durch das Gradientenabstiegsverfahren erreichen [19, S. 56–57].

Gradientenabstiegsverfahen

Das Gradientenabstiegsverfahren dient dazu, Minima von (konvexen) Funktionen zu finden. Dabei wird die Variable der Funktion in Richtung des negativen Gradienten verändert. Dies lässt sich wie folgt auf die Straffunktion in Gleichung (4.4) zur Optimierung der Gewichte anwenden

$$\mathbf{w} := \mathbf{w} + \Delta\mathbf{w} \quad \text{, wobei} \quad \Delta\mathbf{w} = -\eta \nabla J(\mathbf{w}) = \eta \sum_i \left(y^{(i)} - \phi \left(z^{(i)} \right) \right) \mathbf{x}^{(i)} \quad . \tag{4.5}$$

Bei obigem Gradientenabstiegsverfahren wird stets der gesamte Stapel Trainingsobjekte verarbeitet. Dies kann bei großen Datenmengen sehr aufwändig werden. Das stochastische Gradientenabstiegsverfahren optimiert die Gewichte nach jedem Trainingselement

$$\Delta\mathbf{w} = \eta \left(y^{(i)} - \phi \left(z^{(i)} \right) \right) \mathbf{x}^{(i)} \quad . \tag{4.6}$$

Die individuelle Aktualisierung jedes Gewichts erzeugt höheres statistisches Rauschen, durch welches lokale Minima übergangen werden können. Wichtig ist, dass die Trainingsdaten in zufälliger Reihenfolge zur Aktualisierung herangezogen werden. Andernfalls kann eine Periodizität in der Optimierung der Gewichte auftreten. [19, S. 57–59, 66]

Logistische Regression

Die logistische Regression setzt, ähnlich wie das Adaline Verfahren, auf dem Grundmodell des Perzeptrons auf und führt eine neue Aktivierungsfunktion ein. Hierfür dient die Definition eines Wahrscheinlichkeitsverhältnisses $\frac{P}{1-P}$ einer binären Klassifizierung. P ist hier die Wahrscheinlichtkeit eines der zwei möglichen Ereignisse ($y = 0,\ y = 1$), ($1 - P$ entspricht somit der Wahrscheinlichkeit des anderen Ereignisses.) Nach Konvention wird $P(y = 1)$ betrachtet. Es wird

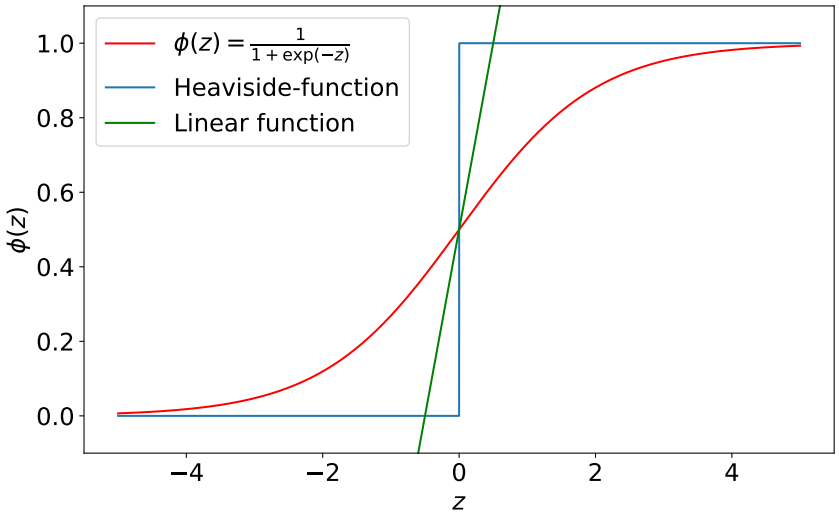


Abbildung 4.2 Der Plot stellt drei Aktivierungsfunktionen dar. Es ist die Heaviside-Funktion, eine logistische Funktion und eine lineare Funktion dargestellt. Die Heaviside-Funktion und die logistische Funktion bilden auf den Wertebereich [0,1] ab, während die lineare Funktion mit einem Offset von 0.5 versehen ist.

weitergehend der natürliche Logarithmus des Wahrscheinlichkeitsverhältnisses verwendet, was die sogenannte logit-Funktion definiert

$$\text{logit}(P) = \ln\left(\frac{P}{1-P}\right) \quad . \tag{4.7}$$

Diese Funktion soll der Nettoeingabe $z = \mathbf{w}^T\mathbf{x}$ eine Klassifizierung zu dem Fall $y = 1$ bei Eingabedaten $\mathbf{x}$ und Gewichten $\mathbf{w}$ wie folgt zuordnen

$$\text{logit}\left(P(y=1|\mathbf{x},\mathbf{w})\right) = z \quad . \tag{4.8}$$

Um die aktuelle Ausgabe des Netzes zu beurteilen ist es hilfreich, die Wahrscheinlichkeit für eine Klassifizierung bei gegebener Eingabe z zu bestimmen. Dies wird mit der Umkehrfunktion, einer sogenannten Sigmoid-Funktion (siehe Abbildung 4.2), erreicht

$$\phi(z) = \frac{1}{1+e^{-z}} \quad . \tag{4.9}$$

$\phi(z)$ ordnet einer Nettoeingabe z die Wahrscheinlichkeit der positiven binären Klassifizierung zu. Zur Optimierung wird die Likelihood der aktuellen Gewichte $\mathbf{w}$ für eine binäre Klassifizierung definiert als

$$L(\mathbf{w}) = P(\mathbf{y}|\mathbf{x},\mathbf{w}) = \prod_{i=1}^{n} P(y^{(i)}|x^{(i)},\mathbf{w}) \tag{4.10}$$

$$= \prod_{i=1}^{n} \left(P(y=1|x^{(i)},\mathbf{w})\right)^{y^{(i)}} \left(P(y=0|x^{(i)},\mathbf{w})\right)^{1-y^{(i)}} \tag{4.11}$$

$$= \prod_{i=1}^{n} \left(\phi(z^{(i)})\right)^{y^{(i)}} \left(1-\phi(z^{(i)})\right)^{1-y^{(i)}} \tag{4.12}$$

Die Exponenten $y^{(i)}$ und $1 - y^{(i)}$ dienen hier der Unterscheidung der zwei Fälle $y = 1$ und $y = 0$. So wird unabhängig vom Zielwert y jeweils die Wahrscheinlichkeit einer Korrektklassifizierung dem Produkt hinzugefügt.
Der Einfachheit halber wird der Logarithmus dieser Funktion betrachtet:

$$l(\mathbf{w}) = \ln\left(L(\mathbf{w})\right) = \sum_{i=1}^{n} y^{(i)} \ln(\phi(z^{(i)})) + (1 - y^{(i)}) \ln(1 - \phi(z^{(i)})) \tag{4.13}$$

Mit $J(\mathbf{w}) = -l(\mathbf{w})$ haben wir eine zu minimierende Straffunktion. Diese Minimierung kann erneut mit dem Gradientenabstiegsverfahren durchgeführt werden. Die Gewichte werden wie folgt aktualisiert [19, S. 81–91]

$$\Delta\mathbf{w} = \eta\nabla l(\mathbf{w}) = -\eta\nabla J(\mathbf{w}) = \eta \sum_{i} \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) \mathbf{x}^{(i)} \quad . \tag{4.14}$$

4.2.3 Mehrschichtige neuronale Netze

Als Weiterführung der Ansätze des Perzeptrons, Adaline und der logistischen Regression lassen sich mehrschichtige neuronale Netze erzeugen [19, 383ff]. Die bisher eingeführten Varianten neuronaler Netze (siehe 4.2.2) umfassen jeweils eine Schicht, in welcher die Eingabewerte gewichtet und mithilfe der Nettoeingabe- und Aktivierungsfunktion zugeordnet werden. Ein (tiefes) neuronales Netz verfügt über mehrere Schichten, in denen jeweils die Rückgabe der Aktivierungsfunktion der vorangegangen Schicht als Eingabe dient. Je Knoten der Folgeschicht existiert ein Satz Gewichte.

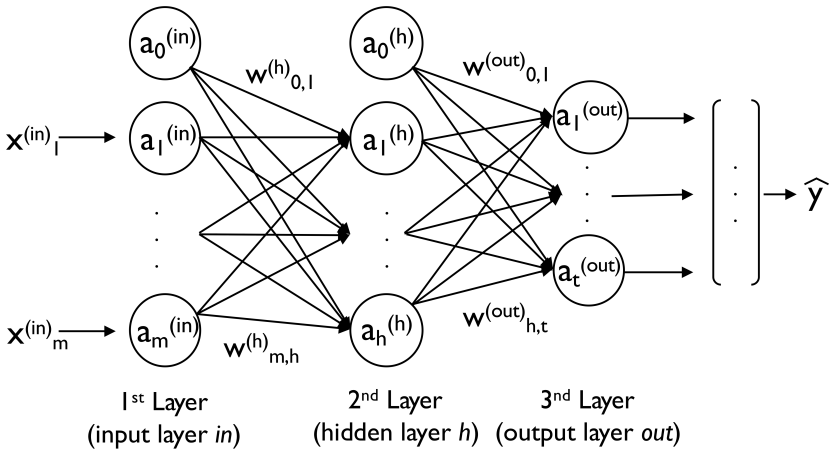


Abbildung 4.3 Schematische Darstellung eines dreischichtigen neuronalen Netzes unter Berücksichtigung der Eingabe $\mathbf{x}$, der Gewichte $\mathbf{w}$ und den Aktivierungen $\mathbf{a}$, welche zur Ausgabe $\hat{y}$ führen. [19, S. 387]

Hierbei werden sogenannte versteckte Schichten (hidden layers) eingefügt. Jede verdeckte Schicht kann über beliebig viele Elemente verfügen. In der Regel ist jedes Element der verdeckten Schicht mit allen Elementen der vorangegangenen Schicht über je ein Gewicht verbunden. In Abbildung 4.3 ist das schematisch dargestellt. In jeder Schicht werden die Nettoeingabe und die Aktivierung berechnet. In der Darstellung wird für jede Schicht eine eigene Bias-Einheit eingeführt.
Die Gewichte zwischen den einzelnen Schichten sind nun Matrizen mit $n \times m$ Elementen, wenn n und m die Anzahl der Elemente der jeweils vorangegangen und folgenden Schicht sind (ohne Bias Einheit der Folge-Schicht) [19, S. 387–392]. Die Berechnung der Ausgabe ist deutlich

aufwändiger und die Anpassung der Gewichte wird nicht-trivial. Auf die genauen Berechnungen wird in Abschnitt 7.3 eingegangen.

4.2.4 Über- und Unteranpassung

Stehen zu viele oder zu wenige Gewichte zur Anpassung zur Verfügung kann es zu Über- und Unteranpassung kommen. Das Modell erreicht entweder eine Komplexität, die nur durch den Trainingsdatensatz gegeben ist oder kann die Komplexität der Gesamtdatenmenge nicht erreichen, da die charakterisierenden Eigenschaften nicht in den gewichteten Merkmalen erfasst werden.
Die Annahme ist, dass hohe Werte der Gewichte aus einer Überanpassung resultieren. Um also die Überanpassung einzuschränken kann die Größe der Gewichte minimiert werden. Hierfür wird die Norm der Gewichte zur Straffunktion addiert. Es wird meist ein Regulierungsparameter λ verwendet, der den Einfluss der Absolutgröße der Gewichte auf die Optimierung steuert. Es ergibt sich die angepasste Straffunktion mit den sogennanten $L1$- und $L2$-Normen zu

$$J'(\mathbf{w}) = J(\mathbf{w}) + \frac{\lambda}{2}\|\mathbf{w}\|_L{}^2 \quad \text{, wobei} \quad \|\mathbf{w}\|_{L2}{}^2 = \mathbf{w}^2 \quad , \quad \|\mathbf{w}\|_{L1} = \sum_{j=1}^{m} |w_j| \quad . \tag{4.15}$$

Es ist Konvention den Kehrwert des Parameters $C = 1/\lambda$ anzugeben.
Durch die $L1$-Regularisierung steigt auch die Wahrscheinlichkeit, das einzelne Gewichte $w_j = 0$ gesetzt werden. So können direkt Merkmale ausgeschlossen werden, die keinen Einfluss auf die Klassifizierung haben.[19, S. 93–95, 141]

4.2.5 Merkmalselektion & -extraktion

Sind die logischen oder physikalischen Zusammenhänge, die zu einer Klassifizierung führen nicht weiter bekannt bzw. sollen nicht in die Architektur des Netzes einfließen, so gibt es verschiedene Verfahren mithilfe von Algorithmen oder anhand der Ausgabe des Netzes selbst die relevanten Merkmale auszuwählen.
Eine der einfachsten Methoden ist es hier, dem Netz unterschiedliche Teilmengen der vorhandenen Merkmale zu Verfügung zu stellen und diese Auswahl anhand der Klassifizierung einer Testdatenmenge zu bewerten. Hier können alle Möglichkeiten durchgegangen werden (sehr rechenaufwändig) oder itereativ anhand der Güte der vorangegangenen Kombinationen einzelne Merkmale hinzugefügt oder entfernt werden.
Ein Ähnliches Vorgehen wird bei sogenannten *Random-Forests* angewendet. Hierbei wird an jedem Knoten des Netzes der Informationsgewinn durch die Verknüpfung bewertet. Dies kann bspw. durch die Entropie geschehen. Nach dieser Analyse können dann einzelne Knoten bzw. Merkmale (wenn die Knoten in der ersten Schicht liegen) entfernt werden [19, S. 147–156].

Bei der Merkmalsextraktion wird die Information der verfügbaren Merkmale auf eine geringere Anzahl Merkmale abgebildet. Spricht man vom Merkmalsraum, welcher durch alle verfügbaren Merkmale aufgespannt wird, so findet eine Abbildung in einen niedrigerdimensionalen Merkmalsraum statt. Hierfür können bspw. lineare Abhängigkeiten zwischen den einzelnen Merkmalen ausgenutzt werden.
Die sogenannte Hauptkomponentenanalyse (Principal Component Analysis - PCA) bestimmt die

Achsen maximaler Varianz im Merkmalsraum. Es lässt sich dann eine Koordinatentransformation durchführen, was den Informationsgehalt der neuen Merkmale maximiert und die Anzahl der Merkmale minimiert [19, S. 159–168].

4.2.6 Hyperparameter

Als Hyperparameter eines neuronalen Netzes bezeichnet man sämtliche Parameter, welche die Klassifizierung beinflussen, allerdings nicht als Teil des Lernprozesses optimiert werden [19, S. 37]. Diese Hyperparameter umfassen beispielsweise die Anzahl der Schichten des Netzes, die einfließenden Merkmale, die Lernrate, die Anzahl der Lernzyklen sowie die Regularisierungsterme.
Zur Optimierung der Hyperparameter werden die Trainingsdaten in mehrere Teilmengen unterteilt. Ein erster Ansatz ist eine einfache Unterteilung in zwei Datensätze, einen Trainings- und einen Testdatensatz. In dieser Arbeit wird je Datensatz ein kleiner Teil ($\leq 1/5$) zur Bewertung der Hyperparameter vorgesehen.
Es gibt verschiedene Verfahren, hier mit k gleichgroßen Teilmengen zu arbeiten, wobei die Anzahl der Teilmengen k der verfügbaren Datenmenge und Rechenkapazität angepasst werden muss. Das Netz wird mit je $k-1$ Teilmengen trainiert, während die k-te Teilmenge zur Bewertung des Netzes dient. Dies wird k-mal durchgeführt, sodass jede Teilmenge einmal zur Bewertung herangezogen wird. Dieses Verfahren nennt sich k-fache Kreuzvalidierung [19, S. 205–210].
Das mit den übrigen Daten trainierte Netz wird auf diese Testdatenmenge angewendet, um eine Über- oder Unteranpassung feststellen zu können. Ein optimal trainiertes Netz erreicht sowohl bei den Test-, als auch bei den Trainingsdaten die gleiche sogenannte Korrektklassifizierungsrate. Diese Rate beschreibt die Anzahl der korrektklassifizierten Elemente der untersuchten Menge. Tritt Über- oder Unteranpassung auf, so werden in der Testdatenmenge weniger korrekte Klassifizierungen getroffen als in der Trainingsdatenmenge.
Abbildung 4.4 zeigt sogenannte Validierungskurven für die verschiedenen Fälle der Über- und Unteranpassung. Es sind jeweils die Korrektklassifizierungsraten (Accuracy) der Test- (Validation) und Trainingsdaten gegenüber die Anzahl der zum Training verwendeten Datensätze (Number of training samples) aufgetragen. Erreichen sowohl Trainings- als auch Testdaten eine schlechte Korrektklassifizierungsrate, so spricht man von einem hohen Bias (High Bias) (zur gewünschten Korrektklassifizierungsrate), was im Plot oben links zu sehen ist. Dies entspricht der Unteranpassung.
Erreicht das Netz für die Trainingsdaten eine hohe Korrektklassifizierungrate, welche jedoch durch die Testdaten bei weitem nicht erreicht wird, so spricht man von einer hohen Varianz (high variance). Dies ist im Plot oben rechts zu sehen und entspricht der Überanpassung des Netzes an die Trainingsdaten.
Der Plot unten rechts zeigt ein optimal trainiertes Netz. Nachdem das Netz mit einer hohen Anzahl Trainingsdaten trainiert wurde, erreicht es für Validierungs- und Trainingsdaten ähnliche Korrektklassifizierungraten [19, S. 211–212].

4.2.7 Bewertung der Klassifizierung

Zur Bewertung des neuronalen Netzes sollten stets Daten zu Rate gezogen werden, welche keinen Einfluss auf die Abstimmung des neuronalen Netzes genommen haben, weder zum Training noch zur Hyperparameterabstimmung [19, S. 205].

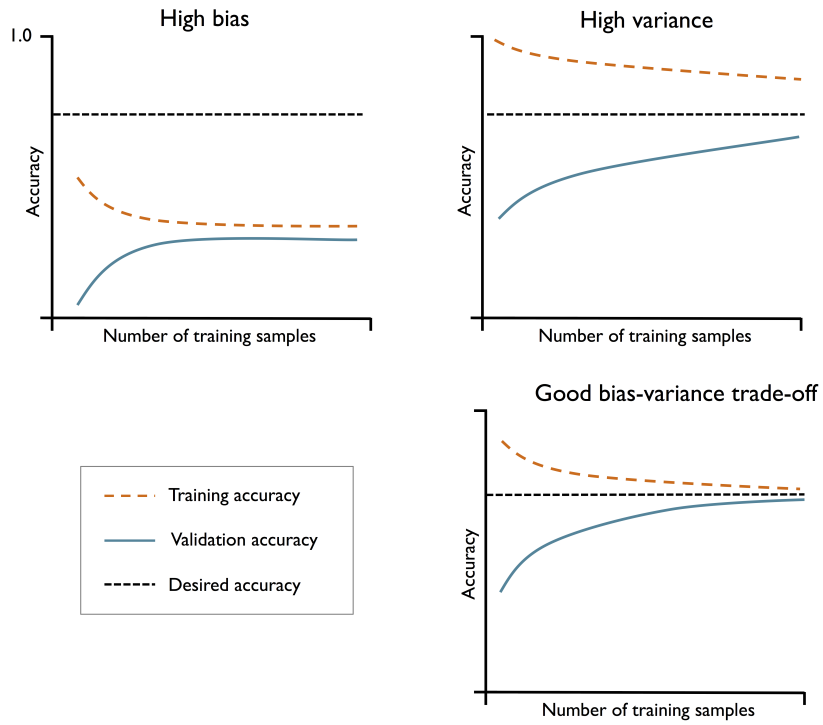


Abbildung 4.4 Darstellung von Unter- und Überanpassung eines lernenden Algorithmus anhand der Korrektklassifizierungsrate (Accuracy) der Trainings- und Testdaten bei unterschiedlich umfangreichen Trainingsdaten. Unten ist das Verhalten eines gut angepassten Netzes zu sehen. [19, S. 212]

Im Folgenden werden die Daten, die zur Bewertung des neuronalen Netzes dienen, Validierungsdaten genannt. Die Validierungsdaten sollten alle Facetten der Datensätze abdecken, welche das neuronale Netz klassifizieren soll. Es ist weiter sinnvoll möglichst unterschiedliche Datensätze zur Validierung verwenden. So kann bewertet werten, ob das Netz nur für eine eingeschränkte Gruppe von Daten verlässliche Ergebnisse liefert.

5 Simulation einer Röntgenanlage mit GATE

Bei der Erstellung der GATE-Skripte wurden sowohl Teile der Beispiele im Wiki [12] als auch der Benchmarks in **Gate/Benchmarks/** [11] übernommen.
In Abbildung 5.1 ist die simulierte Röntgenanalage schematisch dargestellt. Im Folgenden wird die Erstellung der einzelnen Bestandteile, also des Detektors, der Röntgenquelle und des Phantoms erläutert.

Das Gate-Skript (vgl. Anhang A.3.1) lädt zu Beginn eine Alias-Datei (**thirdalias_*.mac**), in welcher sämtliche Parameter in Aliasen gesetzt werden (siehe Anhang A.3.2). Aliase werden in geschweiften Klammern aufgerufen. Dieser Ansatz wurde gewählt, um den Code übersichtlicher zu gestalten und eine einfachere Anpassung zu ermöglichen. Des Weiteren wird zu Beginn eine Materialliste **GateMaterials.db** geladen. Diese entspricht der standardmäßigen Materialbibliothek, verfügbar unter [11]. Die Visualisierung wird in der Regel deaktiviert, andernfalls wird die Visualisierungsdatei **vis2.mac** geladen (siehe Anhang A.3.3).

```
1  ######## EXECUTE & SOURCE EXTERNAL FILES ########
   /control/execute thirdalias_cluster.mac
3  /gate/geometry/setMaterialDatabase GateMaterials.db

5  ######## SET THE VISUALIZATION ########
   /vis/disable
7  #/control/execute vis2.mac
```

5.1 Erstellung eines Detektors

Der Detektor muss als erstes Tochtervolumen des Grundvolumens **world** definiert werden. Daraufhin wird die Position durch die Abweichung vom Mittelpunkt des Grundvolumens definiert (**setTranslation**). Des Weiteren werden die Größe und das Material des Detektors definiert durch **set*Length** und **setMaterial**. Ein Detektor lässt sich weiter unterteilen in **module**, **cluster** und **pixel**. Hier findet lediglich die Unterscheidung in Pixel statt, Modul und Cluster werden dem Gesamtdetektor gleichgesetzt. Die Pixel werden zunächst in Größe und Material definiert und dann mit der entsprechenden Anzahl und einem Richtungsvektor angeordnet.
Um ein detektierfähiges Gerät zu erhalten, müssen die einzelnen Komponenten an das System als detektierfähiges Objekt durch **attach** übergeben werden.

```
1  ######## DEFINE THE DETECTOR ########
   /gate/world/daughters/name CTscanner
3  /gate/world/daughters/insert box        #shape of the detector
   /gate/CTscanner/placement/setTranslation {CTscannerPosX} {CTscannerPosY} {
       CTscannerPosZ} {CTscannerPosSCALE}        #position of the detector
5  /gate/CTscanner/geometry/setXLength {CTscannerSizeX}
   /gate/CTscanner/geometry/setYLength {CTscannerSizeY}
7  /gate/CTscanner/geometry/setZLength {CTscannerSizeZ}
   /gate/CTscanner/setMaterial {CTscannerMaterial}
9  /gate/CTscanner/vis/forceWireframe
   /gate/CTscanner/vis/setColor white
11
```

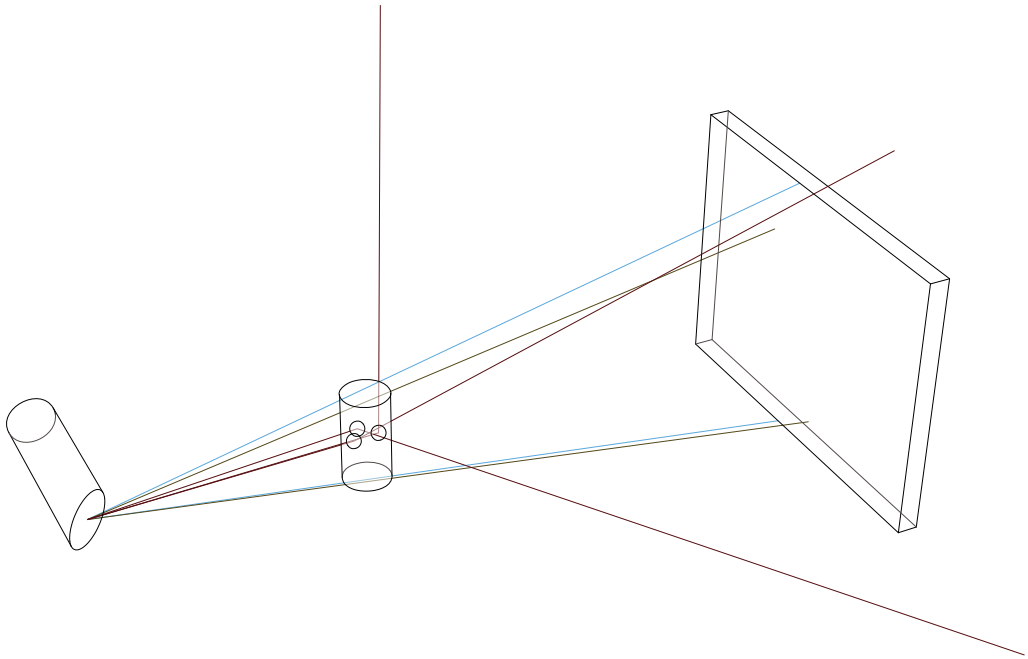


Abbildung 5.1 Schematische Darstellung der simulierten Röntgenanlage. Links in der Abbildung ist die Röntgenquelle zu sehen. In der Mitte der Abbildung ist ein zylinderförmiges Phantom mit kugelförmigen Strukturen dargestellt. Rechts in der Abbildung ist der Detektor skizziert. Es sind Strahlengänge von gestreuten und ungestreuten Photonen eingezeichnet. In blau sind die längsten, den Detektor erreichenden Trajektorien eingezeichnet.

```
######## DEFINE THE MODULES INSIDE THE DETECTOR ########
/gate/CTscanner/daughters/name module       #the module is a daughter volume of the
    detectror
/gate/CTscanner/daughters/insert box
/gate/module/geometry/setXLength {moduleSizeX}
/gate/module/geometry/setYLength {moduleSizeY}
/gate/module/geometry/setZLength {moduleSizeZ}
/gate/module/setMaterial {moduleMaterial}
/gate/module/vis/forceWireframe
/gate/module/vis/setColor white

######## DEFINE THE CLUSTERS INSIDE THE MODULES ########
/gate/module/daughters/name cluster
/gate/module/daughters/insert box
/gate/cluster/geometry/setXLength {clusterSizeX}
/gate/cluster/geometry/setYLength {clusterSizeY}
/gate/cluster/geometry/setZLength {clusterSizeZ}
/gate/cluster/setMaterial {clusterMaterial}
/gate/cluster/vis/forceWireframe
/gate/cluster/vis/setColor white

######## DEFINE THE PIXELS INSIDE THE CLUSTERS ########
/gate/cluster/daughters/name pixel
/gate/cluster/daughters/insert box
/gate/pixel/geometry/setXLength {pixelSizeX}
/gate/pixel/geometry/setYLength {pixelSizeY}
```

```
/gate/pixel/geometry/setZLength {pixelSizeZ}
/gate/pixel/setMaterial {pixelMaterial}
/gate/pixel/vis/forceSolid
/gate/pixel/vis/setColor red

# REPEAT THE PIXElS #
/gate/pixel/repeaters/insert cubicArray
/gate/pixel/cubicArray/setRepeatNumberX {pixelnumberX}      # the number of pixels
    in each dimension
/gate/pixel/cubicArray/setRepeatNumberY {pixelnumberY}
/gate/pixel/cubicArray/setRepeatNumberZ {pixelnumberZ}
/gate/pixel/cubicArray/setRepeatVector {pixelVector}        #the vector to repeat
    the pixels
/gate/pixel/cubicArray/autoCenter true

######## ATTACH SYSTEM #######
/gate/systems/CTscanner/module/attach module
/gate/systems/CTscanner/cluster_0/attach cluster
/gate/systems/CTscanner/pixel_0/attach pixel

######## ATTACH THE DETECTING OBJECTS ########
/gate/pixel/attachCrystalSD
```

5.2 Erstellung einer Röntgenquelle

In der Simulationssoftware GATE kann eine Röntgenröhre auf verschiedene Arten charakterisiert werden [21]. In diesem Fall wird die Röhre durch die geometrischen Eigenschaften des Brennflecks sowie durch ein Energiespektrum definiert.
Die Röhre wird als **xraygun** definiert, welche Photonen (**gamma**) aussendet. Die Größe des Brennflecks wird durch die Aliase **focusHalfSizeX** und **focusHalfSizeY** als je halbe Durchmesser bereitgestellt. Die Position wird in dem Makro **centre** gesetzt. **maxtheta** definiert den Abstrahlwinkel. Da hier das Energiespektrum über eine Verteilung bereitgestellt wird, werden die minimale und die maximale Energie sowie sämtliche bekannte Stützpunkte angegeben.

```
######## SOURCE ########
/gate/source/addSource xraygun      #add the source
/gate/source/verbose 0
/gate/source/xraygun/gps/verbose 0
/gate/source/xraygun/gps/particle gamma     # define the particle
/gate/source/xraygun/gps/type Plane     # define the form
/gate/source/xraygun/gps/shape Rectangle        # define the shape
/gate/source/xraygun/gps/halfx {focusHalfSizeX}     # define the size
/gate/source/xraygun/gps/halfy {focusHalfSizeY}
/gate/source/xraygun/gps/centre {focusPosX} {focusPosY} {focusPosZ} {focusPosSCALE}
        # define the position
/gate/source/xraygun/gps/maxtheta {focusAngle}      # define the emitting angle
/gate/source/xraygun/gps/angtype iso
/gate/source/xraygun/gps/energytype Arb     # define the sources spectrum by
    profiding a histogram
/gate/source/xraygun/gps/histname arb
/gate/source/xraygun/gps/emin 3.00 keV
```

```
/gate/source/xraygun/gps/emax 600.00 keV
/gate/source/xraygun/gps/histpoint   .003 .386377193910233
/gate/source/xraygun/gps/histpoint   .0041639278557114  0.617314450096939
/gate/source/xraygun/gps/histpoint   .0053278557114229  0.893803494023178
/gate/source/xraygun/gps/histpoint   .0065917835671343  1.22408032839862
/gate/source/xraygun/gps/histpoint   .0077557114228457  1.60781684261882
```

Der Großteil des Spektrums wird hier ausgelassen.

```
/gate/source/xraygun/gps/histpoint   .594018036072144  0.028237121886558
/gate/source/xraygun/gps/histpoint   .595214428857716  0.022493538463034
/gate/source/xraygun/gps/histpoint   .596410821643287  0.016798061785091
/gate/source/xraygun/gps/histpoint   .597607214428858  0.011150449572943
/gate/source/xraygun/gps/histpoint   .598803607214429  0.005551163981842
/gate/source/xraygun/gps/histpoint   .600 0

/gate/source/xraygun/gps/arbint Lin
/gate/source/list
```

5.2.1 Simulation von Röntgenspektren

Um eine realistische Röntgenquelle in GATE verwenden zu können, muss ein Spektrum gemessen oder simuliert werden. In dieser Anwendung wird es simuliert. Zur Simulation wird die Software XPECGEN [22] in der Version 1.3.0 mit PYTHON verwendet. Diese Software implementiert ein Modell, welches auf Annahmen über den Elektronenfluss und die Wirkungsquerschnitte im Targetmaterial basiert und an Monte Carlo-simulierte Daten angepasst wurde [23].
Zur Simulation eines Röntgenspektrums werden als Parameter die Beschleunigungsspannung (Energie der Elektronen), das Target-Material sowie der Abstrahlwinkel und der Winkel der Targetoberfläche herangezogen.

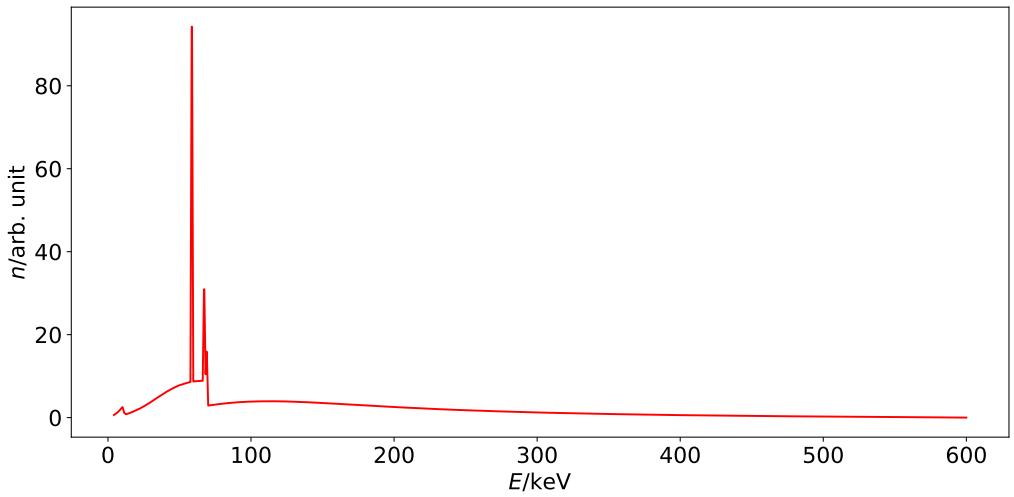


Abbildung 5.2 Simuliertes Röntgenspektrum mit einer Beschleunigungsspannung von 600kV, einer Wolfram Anode und 15° Abstrahlwinkel, 0° geneigter Oberfläche, simuliert mit [22], nach [23].

5.3 Erstellung eines Phantoms

Die Erstellung eines Phantoms in GATE erfolgt zunächst über die geometrische Definition verschiedener Objekte. Durch ein mother-daughter-System können Objekte ineinander erstellt und global Eigenschaften (bspw. eine Rotation) zugewiesen bekommen.
Das Phantom wird zunächst als Tochtervolumen der **world** initialisiert. Daraufhin wird der Körper durch eine Grundform angegeben (bspw.: **cylinder**) und im Weiteren durch seine speziellen geometrischen Eigenschaften beschrieben. Mit **setMaterial** wird ein Material zugewiesen. **setColor** dient lediglich zur Farbgebung in der Visualisierung. Es können mit **/moves/rotation** verschiedene Arten von Bewegungen für ein Objekt eingefügt werden. Bewegt sich ein Objekt, werden alle Tochtervolumen mitbewegt.
Es werden diverse Objekte verschiedener Materialien auf dieselbe Art und Weise erstellt. Der folgende Code stellt die Erstellung eines rotierenden Zylinders dar. Innerhalb des Zylinders befinden sich mehrere Ebenen radial angeordneter Kugeln sowie eine Kugel im Zentrum. Das restliche Volumen des Zylinders ist mit Wasser gefüllt.

```
1  ######## DEFINE THE PHANTOM ########
   /gate/world/daughters/name waterCylinder        #name of the first part of the
       phantom
3  /gate/world/daughters/insert cylinder       #shape
   /gate/waterCylinder/placement/setRotationAxis 1 0 0     #define the placement
5  /gate/waterCylinder/placement/setRotationAngle 90. deg
   /gate/waterCylinder/geometry/setRmin 0. mm      #geometry definition
7  /gate/waterCylinder/geometry/setRmax {CylinderRadius}
   /gate/waterCylinder/geometry/setHeight {CylinderHeight}
9  /gate/waterCylinder/setMaterial {CylinderMaterial}
   /gate/waterCylinder/vis/forceWireframe
11 /gate/waterCylinder/vis/setColor cyan

13 /gate/waterCylinder/moves/insert rotation       #rotate the WaterCylinder and all
       its daughter volumes
   /gate/waterCylinder/rotation/setSpeed {WaterCylinderRotation}       #the degrees of
       rotation per slice
15 /gate/waterCylinder/rotation/setAxis 0 0 1

17 /gate/waterCylinder/daughters/name Ball_mid
   /gate/waterCylinder/daughters/insert sphere
19 /gate/Ball_mid/placement/setTranslation {Ball_mid_Pos}
   /gate/Ball_mid/geometry/setRmin 0. mm
21 /gate/Ball_mid/geometry/setRmax {Ball_mid_Radius}
   /gate/Ball_mid/setMaterial {Ball_mid_Material}
23 /gate/Ball_mid/vis/forceSolid
   /gate/Ball_mid/vis/setColor white
25
   /gate/waterCylinder/daughters/name Ball_1
27 /gate/waterCylinder/daughters/insert sphere
   /gate/Ball_1/placement/setTranslation {Ball_1_Pos}
29 /gate/Ball_1/geometry/setRmin 0. mm
   /gate/Ball_1/geometry/setRmax {Ball_1_Radius}
31 /gate/Ball_1/setMaterial {Ball_1_Material}
   /gate/Ball_1/vis/forceSolid
```

```
/gate/Ball_1/vis/setColor yellow
```

Es werden fünf weitere Kugeln auf dieselbe Weise erstellt.

```
# REPEAT PARTS OF THE PHANTOM #
/gate/Ball_1/repeaters/insert linear        # repeat all the balls
/gate/Ball_1/linear/setRepeatVector {BallsVector}
```

Die fünf weiteren Kugeln werden ebenso wiederholt.
Zusätzlich zu den definierten Kugeln innerhalb des Zylinders werden Buchstaben als dreidimensionale Objekte (das griechische Φ) in einer Matrix außerhalb des Zylinders, parallel zur Detektorebene, angeordnet. Diese Φ-Matrix befindet sich in einem Quader, welcher ebenfalls mit Wasser gefüllt ist. Die Matrix wird in diesem Fall in jeder Projektion um 180° gedreht, sodass sie sich stets parallel zum Detektor, aber einmal vor und einmal hinter dem Zyinder befinden.
Die Φ-Matrix wird dem Phantom hinzugefügt, um zusätzlich zu den groben Strukturen der Kugeln, noch feine Strukturen zu schaffen. Die Größe der Strukturen der Buchstaben entsprechen je nach Drehwinkel bei der Aufnahme nahezu der Pixelgröße.

```
/gate/world/daughters/name PhiBox
/gate/world/daughters/insert box
/gate/PhiBox/placement/setTranslation {PhiBoxPosX} {PhiBoxPosY} {PhiBoxPosZ} {
    PhiBoxPosSCALE}
/gate/PhiBox/geometry/setXLength {PhiBoxSizeX}
/gate/PhiBox/geometry/setYLength {PhiBoxSizeY}
/gate/PhiBox/geometry/setZLength {PhiBoxSizeZ}
/gate/PhiBox/setMaterial {PhiBoxMaterial}
/gate/PhiBox/vis/forceWireframe
/gate/PhiBox/vis/setColor cyan

/gate/PhiBox/daughters/name Phi
/gate/PhiBox/daughters/insert tessellated       # include a stl file as phantom
/gate/Phi/placement/setTranslation {PhiPos}
/gate/Phi/geometry/setPathToSTLFile {PhiPath}
/gate/Phi/setMaterial {PhiMaterial}

/gate/Phi/repeaters/insert cubicArray       # repeat the stl
/gate/Phi/cubicArray/setRepeatNumberX {PhinumberX}
/gate/Phi/cubicArray/setRepeatNumberY {PhinumberY}
/gate/Phi/cubicArray/setRepeatNumberZ {PhinumberZ}
/gate/Phi/cubicArray/setRepeatVector {PhiVector}
/gate/Phi/cubicArray/autoCenter true


/gate/PhiBox/moves/insert orbiting      # rotate the PhiBox
/gate/PhiBox/orbiting/setSpeed {PhiBoxRotation}
/gate/PhiBox/orbiting/setPoint1 0 1 0 cm
/gate/PhiBox/orbiting/setPoint2 0 0 0 cm
```

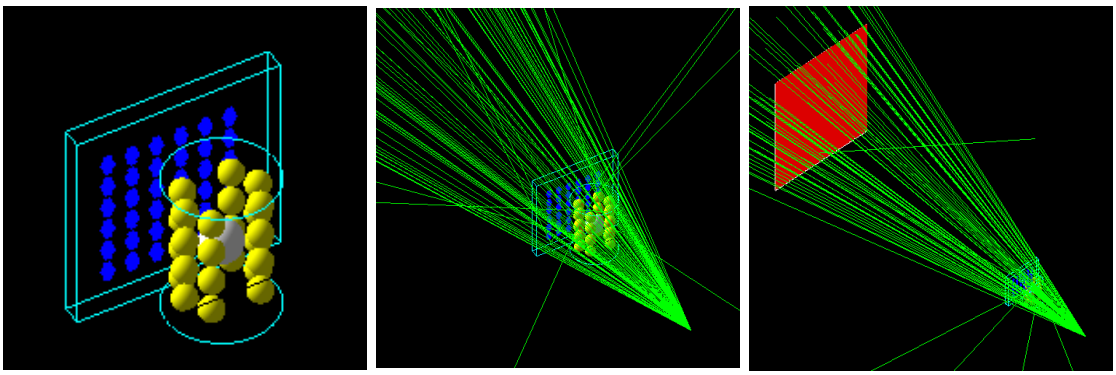
Durch die **attach**-Befehle werden die Elemente des Phantoms in die Simulation aufgenommen.

```
######## ATTACH THE PHANTOM ########
/gate/waterCylinder/attachPhantomSD
/gate/Ball_mid/attachPhantomSD
/gate/Ball_1/attachPhantomSD
/gate/Ball_2/attachPhantomSD
/gate/Ball_3/attachPhantomSD
```

```
/gate/Ball_4/attachPhantomSD
/gate/Ball_5/attachPhantomSD
/gate/Ball_6/attachPhantomSD
/gate/PhiBox/attachPhantomSD
/gate/Phi/attachPhantomSD
```

5.4 Durchgeführte Simulationen

In Abbildung 5.3a ist der Aufbau des Phantoms mit dem die Trainingdaten erstellt werden zu sehen. In gelb sind die zylindrisch angeordneten Kugeln dargestellt und in weiß die mittlere Kugel, in dessen Mitte sich die Drehachse befindet. In blau sind die dreidimensionalen Buchstaben Φ zu erkennen. Die Anordnung im Zylinder rotiert, um viele verschiedene Projektionen zu erzeugen. Die Buchstabenmatrix rotiert ebenfalls, allerdings mit einer höheren Geschwindigkeit.
In Abbildung 5.3b sind zusätzlich zum Phantom die simulierten Trajektorien einiger Photonen zu sehen. Hierdurch wird die Position der Röntgenquelle sichtbar. Es ist auch zu erkennen, dass einige Photonen im Objekt gestreut werden. In Abbildung 5.3c ist noch der Detektor mit ins Bild genommen. Es ist erkennbar, dass der Abstrahlwinkel der Röntgenquelle einen größeren Raumwinkel abdeckt als der Detektor. Des Weiteren fällt auf, dass auch im Detektor selbst Streuung stattfindet.

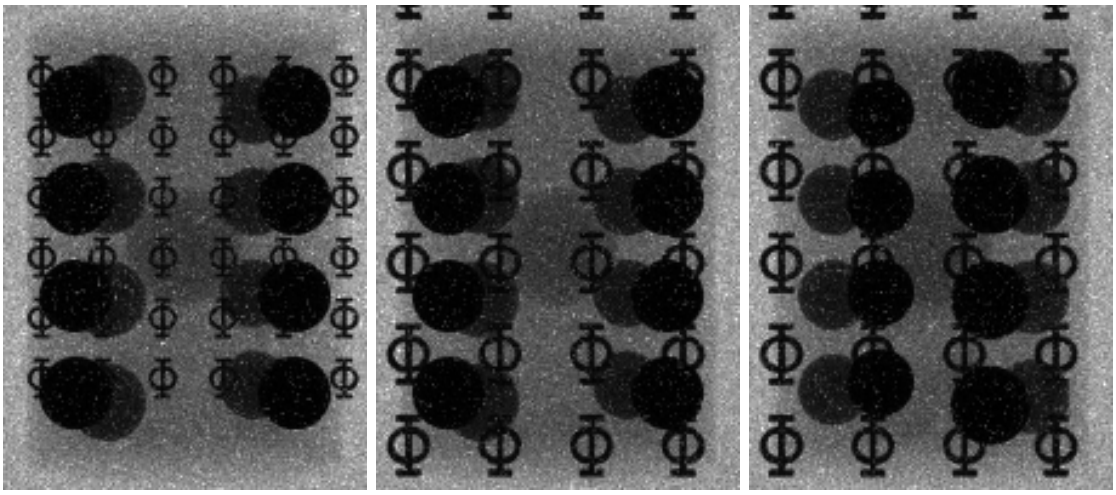


(a) (b) (c)

Abbildung 5.3 Aufbau des Phantoms in der Simulation. In gelb und weiß sind die Kugeln im Zylinder zu erkennen, in blau die Buchstaben. Abbildungen (b) und (c) beeinhalten Trajektorien von Photonen.

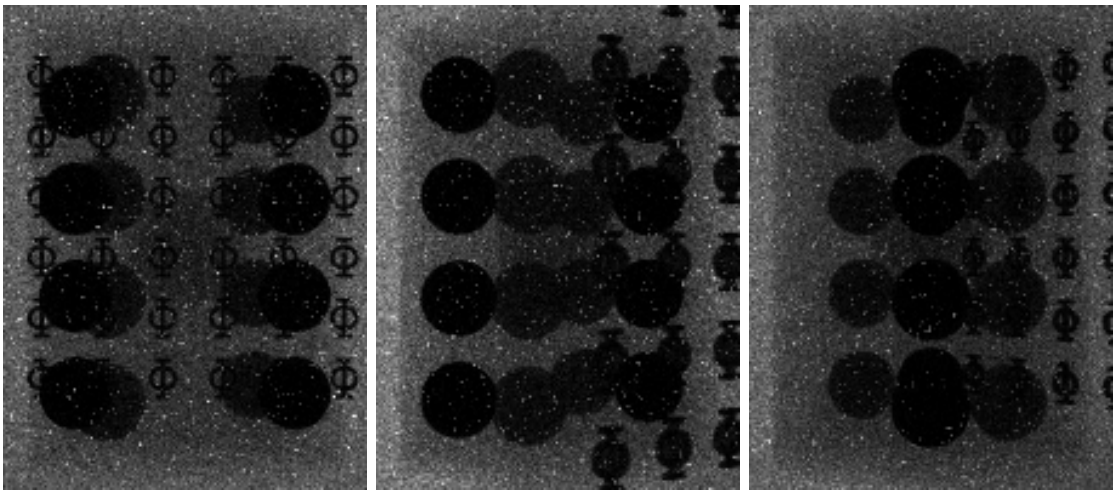
5.4.1 Trainingsdaten

In Abbildung 5.4 sind drei Röntgenprojektionen eines Phantoms (I) zu sehen. In jeder Projektion ist die Φ-Matrix parallel zum Detektor ausgerichtet. In 5.4a ist die Matrix zwischen Zylinder und Detektor, während sie in 5.4b und 5.4c zwischen Röntgenquelle und Zylinder steht. Der Zylinder ist in jeder Projektion anders orientiert. Die überlagerte Streustrahlung ist bereits zu erkennen. Die Aufnahmen werden mit 3333333 Photonen je Projektion erstellt (Datensatz II stellt hier eine Ausnahme dar). Das unterliegende Röntgenspektrum umfasst Energien zwischen 3 keV und 600 keV und ist in Abbildung 5.2 (Abschnitt 5.2.1) dargestellt. Ansonsten gelten die in den Abschnitten 5.1, 5.2 und 5.3 erörterten Merkmale der Simulation.
Die ersten beiden Datensätze (I, II) werden mit einem Phantom mit Kugeln aus Wolfram, Aluminium, Knochen und Germanium erzeugt. Die verschiedenen Materialien wurden ausgewählt,



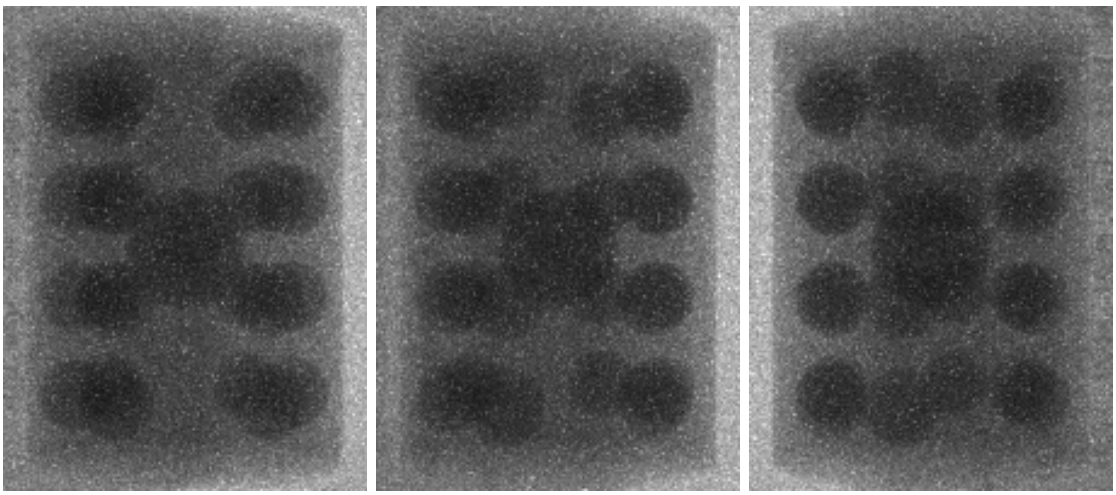
(a) (b) (c)

Abbildung 5.4 Simulierte Röntgenprojektionen des ersten erstellten Phantoms I. Ein Röntgenspektrum mit einer maximalen Energie von 600keV bildet einen Zylinder gefüllt mit Materialien zwischen Knochen und Wolfram aus verschiedenen Richtungen ab.



(a) (b) (c)

Abbildung 5.5 Simulierte Röntgenprojektionen des Phantoms II. Ein Röntgenspektrum mit einer maximalen Energie von 600keV bildet einen Zylinder gefüllt mit Materialien zwischen Knochen und Wolfram aus verschiedenen Richtungen ab.



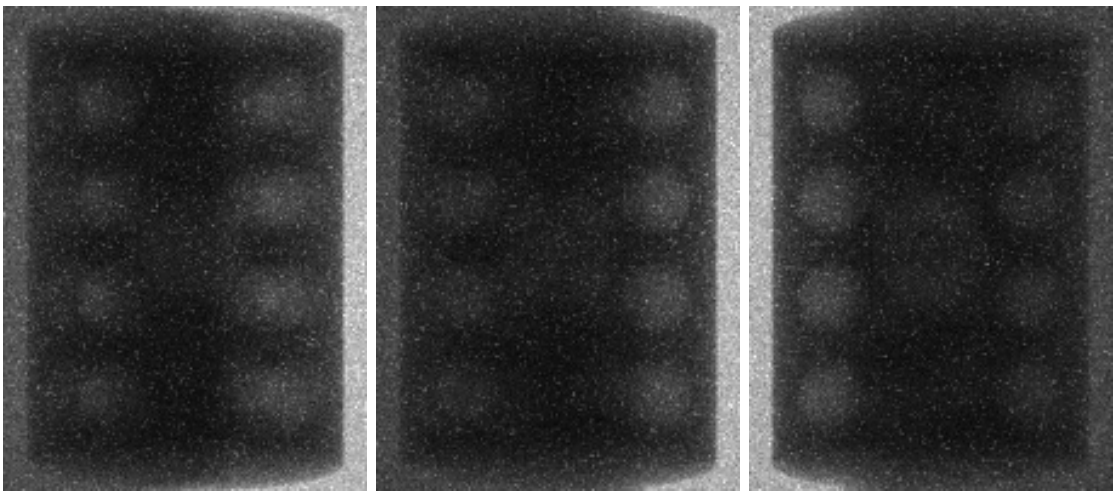
(a) (b) (c)

Abbildung 5.6 Simulierte Röntgenprojektionen des ersten erstellten Phantoms III. Ein Röntgenspektrum mit einer maximalen Energie von 600keV bildet einen Zylinder, welcher Aluminiumkugeln enthält, ab.

um eine möglichst große Bandbreite abzudecken und die Schwierigkeiten der gleichzeitigen Auflösung verschiedener Materialien herauszustellen. Die Buchstaben Φ bestehen aus Wolfram. Die beiden Datensätze unterscheiden sich lediglich in der Projektionszahl. Der erste Datensatz I besteht aus 30 Projektionen, die unter 30, in 720° gleichverteilten Winkeln des Zylinders aufgenommen werden. Hierbei wird die Φ-Matrix je Projektion um 180° gedreht, sodass es keine identischen Projektionen über die zwei Zylinderrotationen gibt. Für den zweiten Datensatz II werden 120 Projektionen, gleichverteilt über 720° aufgenommen. Hier sind abweichend zu den anderen Simulationen lediglich 833333 Photonen je Projektion simuliert. Hierbei wird die Φ-Matrix je Projektion um 45° gedreht, was ebenfalls dafür sorgt, dass keine identischen Projektionen erzeugt werden. In Abbildung 5.4 sind drei Projektionen des Datensatzes I zu sehen. Abbildung 5.5 zeigt Projektionen des Datensatzes II. Die Projektionen des Datensatzes II unterscheiden sich durch die Objektposition und die Belichtungszeit.
Es werden zwei weitere Datensätze (III, IV) erstellt. Bei beiden werden wieder 120 verschiedene Projektionen erstellt. Alle Bauteile bestehen in diesem Fall aus Aluminium. Geometrish bleibt das Phantom bei der ersten Simulation das gleiche, bei der zweiten Simulation wird es invertiert, dass heißt der Zylinder besteht bis auf die Kugeln aus Alumnium. Selbiges gilt für den Quader, welcher die Φ's enthält. Die Kugeln und Φ's werden mit Luft gefüllt. Projektionen dieser Zylinder sind in den Abbildungen 5.6 (III) und 5.7 (IV) zu sehen. Es fällt auf, dass die Φ-Matrix im Vergleich zu Abbildung 5.4 nicht zu erkennen ist. Ihre Position ist durch eine Verdunkelung im entsprechenden Bildbereich zu erkennen. Dies liegt daran, dass diese nun ebenfalls aus Alumnium besteht. In den Phantomen I und II bestanden die Buchstaben aus Wolfram, welches eine deutlich höhere Dichte und Ordnungszahl besitzt.

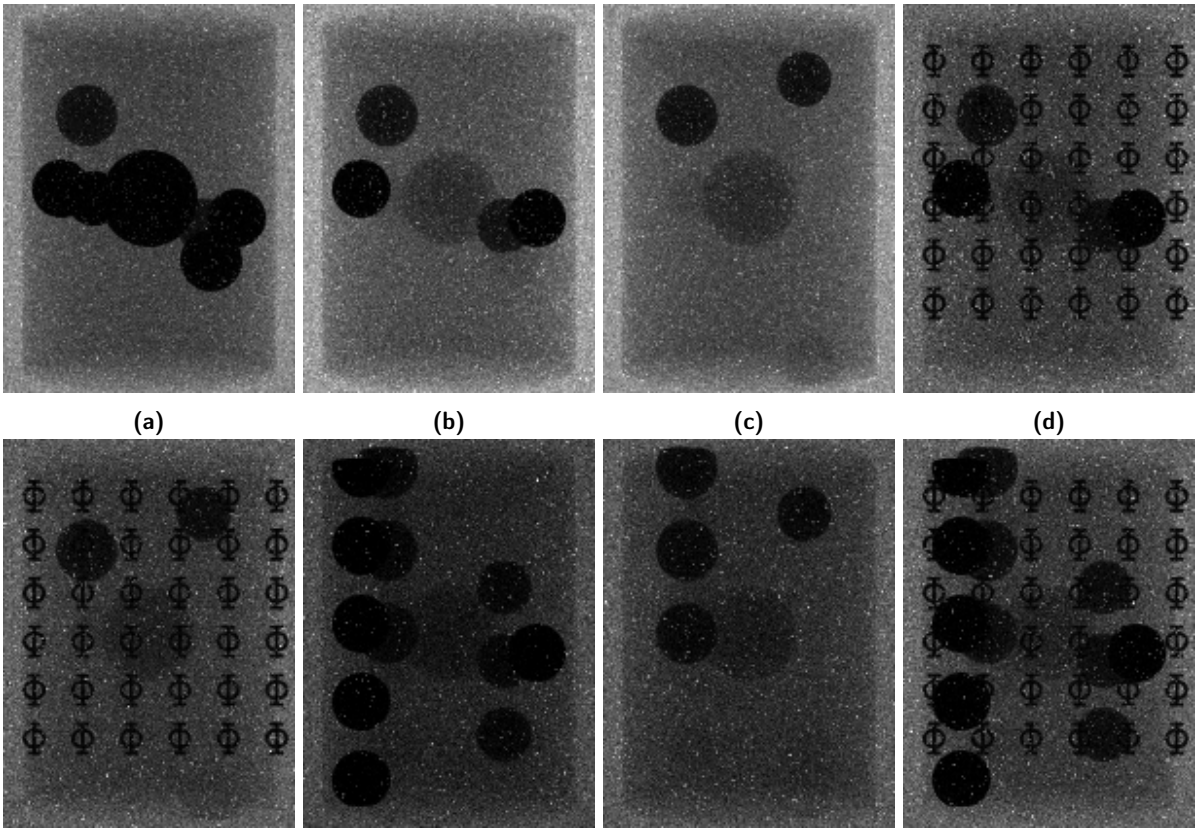
5.4.2 Validierungsdaten

Um Validierungsdaten zu erzeugen wird mit geometrisch abweichenden Phantomen gearbeitet. In Abbildung 5.8 sind diese acht Phantome dargestellt. Sämtliche Validierungsphantome bestehen



(a) (b) (c)

Abbildung 5.7 Simulierte Röntgenprojektionen des ersten erstellten Phantoms IV. Ein Röntgenspektrum mit einer maximalen Energie von 600keV bildet einen Aluminiumzylinder mit kugelförmigen Löchern aus verschiedenen Richtungen ab.



(a) (b) (c) (d)

(e) (f) (g) (h)

Abbildung 5.8 Simulierte Röntgenprojektionen der Validierungsphantome, simuliert mit GATE.

aus verschiedenen Materialien. Diese umfassen, ähnlich wie bei dem Phantom der Datensätze I und II Wolfram, Blei, Aluminium, Knochen und Germanium. Bei drei Phantomen ist die Φ-Matrix aus Wolfram enthalten.

6 Datenverarbeitung

Im Verlauf der Datenerzeugung und Analyse wird mit verschiedenen Datentypen und Strukturen umgegangen. In diesem Abschnitt werden die vorkommenden Datentypen, Schnittstellen und Verarbeitungsschritte erläutert.

6.1 Ausgabe durch GATE

GATE bietet die Möglichkeit, Simulationsdaten auf verschiedene Arten und Weisen auszugeben. In den Simulationen dieser Arbeit werden die entstehenden Projektionsdaten als Binärdaten ausgegeben und die weiteren Simulationsdaten in eine ROOT Datei geschrieben.

6.1.1 Projektionen als Binärdateien

Die in Binärdateien gespeicherten Projektionen umfassen Graustufenbilder, welche in little-endian byte-order als 32bit Floating-Point Zahlen gespeichert werden. Die folgende Funktion zeigt, wie diese Dateien mit PYTHON in einen **numpy**-Array gelesen werden können. Dieser Array kann dann als Bild ausgegeben werden (vgl. das vollständige Skript A.2.1). Mit den Projektionen findet keine weitere Datenverarbeitung statt, sie dienen lediglich dazu, auf einfachem Weg das Ergebnis einer Simulation abschätzen bzw. überprüfen zu können.

```
def read_image(file, columns, rows):
  data = np.fromfile(file, '<f4')  # little-endian float32
  data.shape = (columns, rows)
  return data
```

6.1.2 ROOT-Tree

Die von GATE ausgegebene ROOT-Datei enthält sämtliche aufgezeichneten Simulationsinformationen. Diese sind in zwei Trees unterteilt, Hits und Singles. Unter Hits sind sämtliche den Detektor treffenden Teilchen hinterlegt. Singles speichert alle unter Berücksichtigung der Ausleseelektronik und weiteren Detektoreigenschaften detektierten Teilchen.
In einem solchen ROOT-Tree sind alle Informationen in sogenannten Branches erfasst. Beispielsweise existiert ein Branch namens **eventID**, der jedem Teilchen im Tree eine individuelle Nummer zuordnet. Unter **pixelID** ist für jedes Teilchen das Pixel gespeichert, in welchem es detektiert wurde. Die im Verlauf der Untersuchungen der Daten verwendeten Branches sind **trackLength** als zurückgelegte Weglänge, **edep** als Energie der Teilchen, **nPhantomCompton** und **nCrystalCompton** als Anzahl der je Teilchen stattgefundenen Compton-Streuungen im Phantom bzw. Detektor-Kristall sowie **nPhantomRayleigh** und **nCrystalRayleigh** als selbige Informationen bezüglich Rayleigh Streuung.
Es lassen sich also detaillierte Informationen, teilchen- und pixelgenau aufgelöst, über verschiedene Streuprozesse, Energien und weitere Parameter erhalten.

6.2 Verarbeitung mit python

Sämtliche Programme zur Datenverarbeitung und Analyse wurden in PYTHON erstellt. Alle Skripte sind mit Version 2.7.15 kompatibel und sollten durch minimale Änderungen auch mit Versionen 3.xx.xx kompatibel sein.
Es werden verschiedene weitere Bibliotheken genutzt. Für weiterführende Rechnungen werden NUMPY 1.15.1 und SCIPY 1.1.0, zur Dateneinlese wird PYROOT/ROOT 6.16.0 und zur graphischen Datenausgabe werden MATPLOTLIB 2.2.3 und PILLOW 5.3.0 verwendet [24, 25, 26, 27, 28]. In einigen Skripten werden für Operation am System noch die Standardbibliotheken OS und SYS eingebunden. Die je Skript verwendeten Bibliotheken werden stets zu Beginn des Skriptes eingebunden. Im Anhang sind sämtliche Skripte in Abschnitt A.2 einsehbar.

6.2.1 Nutzung einer Schnittstelle zwischen ROOT und python

Das Paket PYROOT dient dazu, ROOT Klassen und Bibliotheken in PYTHON zu nutzen [26]. Über diese Schnittstelle kann der durch GATE erstellte ROOT-Tree weiterverarbeitet und analysiert werden.
Das erstellte Programm kann mehrere erstellte Projektionen einer GATE-Simulation gleichzeitig verarbeiten. In dem folgenden Code (gesamtes Skript, siehe Anhang A.2.2) werden zunächst die Ein- und Ausgabgeverzeichnisse definiert, um dann mit dem Befehl `Tree = Treefile.Get(ROOT.TFile(rootpath))` die Simulationsdaten in das Objekt `Tree` zu öffnen. Für alle zu extrahierenden Elemente des ROOT Trees werden Elemente der Klasse `data_matrices` in Form einer Liste angelegt (vgl. hierzu Abschnitt 6.2.2). Der Befehl `Tree.GetEntries()` lädt die Anzahl der Einträge des Trees und `Tree.GetEntry(i)` lädt den Eintrag `i`. Das Element `Tree.time` ist der Zeitstempel des Eintrags, welcher dazu dient, mit der `if`-Abfrage `if (Tree.time >j+1)& (Tree.time <j+2):` den Eintrag der Projektion `j` zuzuordnen. Daraufhin wird der Eintrag an das `data_matrices`-Objekt über die `update`-Funktion übergeben.

```
6  ######## DEFINE INPUT/OUTPUT PARAMETERS ########
   columns = 150
8  rows = 200
   base_inputdir = '/run/media/sj/SJ1/Bachelor/Data_sjung/simulations_5/
                                          thirdCT_400000000_120slices_'
10 numinputdirs = 50
   inputdirs = []
12 base_base_outputdir = '/run/media/sj/SJ1/Bachelor/Data_sjung/simulations_5/
                                          thirdCT_400000000_120slices_'
   base_outputdir = '/run/media/sj/SJ1/Bachelor/Data_sjung/simulations_5/
                                          thirdCT_400000000_120slices_10/arrays'
14 numoutputdirs = 120

16 for j in range(0,numinputdirs):   #make input directories
     inputdirs.append(base_inputdir + str(j)+'/')
18
   for inputdir in inputdirs:
20   outputdirs = []
     for j in range(0, numoutputdirs):   #make output directories
22     outputdirs.append(inputdir + 'arrays' + str(j)+'/')
       if not os.path.exists(outputdirs[j]):
```

```
24        os.mkdir(outputdirs[j], 0755)
    print inputdir
26
###### READ ROOT PATH #######
28    rootpath = inputdir + "ct_results.root"
    Treename = "Hits"
30    Treefile = ROOT.TFile(rootpath)
    Tree = Treefile.Get(Treename)
32    n_Tree = Tree.GetEntries()

34    time = numoutputdirs    #the number of time slices in the simulation
    timeslices = np.arange(0,time+1)
36    numfeatures = 1   #the number of features to extract
    trackLength = 0   #define the feature to extract
38    features = [[data_matrices(columns*rows, columns, rows) for _ in range(0,
                                        numfeatures)] for _ in range(0,time)]
    featurestrings = ['trackLength']
40    print n_Tree
    j = 0
42
######## EXTRACT THE SELECTED DATA FROM THE ROOT TREE #######
44    for i in range(int(n_Tree/1)):
      Tree.GetEntry(i)    #read the  ith Entry in the root tree
46      if (Tree.time > j+1) & (Tree.time < j+2) & (Tree.time < time):    #discriminate
                                        the timeslice the entry belongs to. last
                                        condition is because seldom there are one
                                        or two counts counted after the actual
                                        simulation time
        j += 1
48        features[j][trackLength].update(Tree, i, Tree.trackLength)
      elif (Tree.time > j) & (Tree.time < j+1):
50        features[j][trackLength].update(Tree, i, Tree.trackLength)
      else:
52        print('ERROR - Unknown Time: ', Tree.time)
######## MAKE DIFFERENT FORMATS OF THE DATA AND SAVE THE SELECTED DATA #######
54    for k in range(0, time):
      print outputdirs[k]
56      for l in range(0, numfeatures):
        features[k][l].reshape(columns, rows)
58        features[k][l].makeprojection(columns, rows)
        features[k][l].savearrays(featurestrings[l], outputdirs[k])
```

6.2.2 Eine Klasse zur Verarbeitung und Speicherung der Simulationsdaten in python

Die `data_matrices`-Klasse umfasst einige Funktionen zur Umwandlung und Speicherung der je Merkmal gesammelten Daten (gesamte Klasse, siehe A.2.3). Ein Objekt wird lediglich mit der Bildgröße initialisiert. In der Initialisierung (`__init__()`) werden sechs Arrays initialisiert. `self.vector` dient als simpelste Speicherform der pixelzugeordneten Werte eines Merkmals aus dem ROOT-Tree. Er wird als eindimensionaler Array mit Nullen initialisiert. Die Größe des Arrays entspricht zunächst der Pixelanzahl, wird aber durch die Funktion `update` auf ganzzahlige

Vielfache dessen erhöht. `self.IDsPerPixel` dient dazu, zu zählen, wie viele Instanzen eines Merkmals je Pixel bereits gespeichert sind. `self.IDsPerPixel_matrix` formt den eindimensionalen Array `self.IDsPerPixel` auf die Projektionsdimensionen um und stellt somit das aus einem Merkmal resultierende Graustufenbild dar. Beide Arrays werden zunächst mit Nullen initialisiert. `self.matrix` bildet eine dreidimensionale Darstellung von `self.vector`. Hier stellen die Bilddimensionen die ersten beiden Größen dar, während die dritte Dimension aus der maximalen Anzahl der Merkmalseinträge pro Pixel folgt. Zunächst wird auch diese Matrix mit Nullen in den Bilddimensionen initialisiert. Zu `self.IDsPerPixel_matrix` wird noch ein Platzhalter für eine fouriertransformierte Matrix `self.IDsPerPixel_fft_matrix` erstellt. Zum Speichern eines Histogrammes der Merkmale dient die zunächst leere Liste `self.hist`.

```
class data_matrices(object):
  """

  Parameters
  ----------
  pixels : int
    number of pixels = columns*rows
  columns : int
    number of columns in a projection
  rows : int
    number of rows in a projection
  IDsPerPixel : array, shape = [rows*columns]
    array containing the number of events in each pixel
  vector : array, shape = [n*pixels]
    the events, mapped to each pixel, n vectors appended to avoid double counting in
                                            one pixel
  matrix : array, shape = [n, rows, columns]
    the vectors, reshaped to a three dimensional matrix, which has the detector
                                          dimensions as two dimensions and the
                                          events per pixel in the third dimension
  IDsPerPixel_matrix : array, shape = [rows, columns]
    a matrix of the detector dimensions, whose elements are the number of counted
                                          events in the corresponding pixel
  hist : the histogram showing the distribution of the values in all pixels
  """
  def __init__(self, pixels, columns, rows, vector = [], matrix = [], IDsPerPixel =
                                          [], IDsPerPixel_matrix = []):
    self.pixels = columns*rows
    self.columns = columns
    self.rows = rows

    self.IDsPerPixel = np.zeros((pixels))
    self.vector = np.zeros((pixels))

    self.IDsPerPixel_matrix = np.zeros((rows, columns))
    self.matrix = np.zeros((1, rows, columns))
    self.IDsPerPixel_fft_matrix = np.zeros((rows, columns))
    self.hist = []
```

Zur Übergabe der Daten eines ROOT-Trees an ein data_matrices-Objekt dient die update-Funktion. Die update-Funktion lädt den Eintrag i aus dem ROOT-Tree (Tree.GetEntry(i)) und speichert den Wert des der Funktion übergebenen Tree_objects in dem self.vector -Array des data_matrices-Objekts an der Stelle int(Tree.pixelID +self.IDsPerPixel[int(Tree.pixelID)]*self.pixels) ab. Der Offset self.IDsPerPixel[int(Tree.pixelID)]*self.pixels wird durch die Werte in dem self.IDsPerPixel-Vektor definiert. Dieser Vektor wurde in der __init__()-Funktion zunächst mit Nullen der Projektionsgröße initialisiert. Nachdem der Wert des Tree_objects gespeichert ist, wird der der int(Tree.pixelID) entsprechende Eintrag im self.IDsPerPixel-Vektor um 1 erhöht. Wenn der Wert des Eintrags der Anzahl der in self.vector speicherbaren Projektionen entspricht, wird self.vector um die Pixelanzahl einer Projektion erhöht. Dies dient dazu, dass in self.vector jeder Wert der Tree_objects einzeln abgelegt werden kann.

```
  def update(self, Tree, i, Tree_object):
40  """Get an entry from the ROOT-Tree and log it to the corresponding pixel in the
                                        vector. Increment the IDsPerPixel counter
                                        and introduce new layer if necessary

42  Parameters
    ----------
44  Tree : ROOT-TFile
      The Tree containing the branches
46  i : int
      the iterator, specifiying which element to read the branches from
48  Tree_object : KORRR
      The selected branch
50
    Returns
52  -------
    self
54  """
    Tree.GetEntry(i)
56  newlayer = np.zeros((self.pixels))
    self.vector[int(Tree.pixelID + self.IDsPerPixel[int(Tree.pixelID)]*self.pixels)]
                                          = Tree_object
58  self.IDsPerPixel[int(Tree.pixelID)] +=1
    if (self.IDsPerPixel[int(Tree.pixelID)] * self.pixels >= len(self.vector)):
60    self.vector = np.append(self.vector, newlayer)
```

Die reshape()-Funktion dient dazu, aus den in self.vector gespeicherten Daten eine dreidimensinoale Matrix self.matrix zu erstellen. Diese Matrix hat als Dimensionen die Form der Projektion und die Anzahl der nötigen Schichten, um alle Einträge des Tree_objects je Pixel speichern zu können.
Eine makeprojection()-Funktion dient dazu, aus der self.matrix eine Projektion zu erstellen, welche als Grauwerte die Anzahl der Einträge des jeweiligen Merkmals je Pixel enthält. Es handelt sich somit um eine Matrix, die alle Einträge der Röntgenprojektion enthält, die dem Merkmal des Tree_objects entsprechen. Diese Matrix wird unter self.IDsPerPixel_matrix gespeichert.

```
62  def reshape(self, columns, rows):
      """Reshape the one-dimensional vector to the three-dimensional matrix.
64
      Parameters
66    ----------
      columns : int
68      number of columns in a projection
      rows : int
70      number of rows in a projection

72    Returns
      -------
74    self
      """
76    var = len(self.vector)/(columns*rows)
      self.matrix = np.reshape(self.vector, (int(var), rows, columns))
78
  def makeprojection(self, columns, rows):
80    """Count the IDs referring to the same pixel and write them in
                                          IDsPerPixel_matrix

82    Parameters
      ----------
84    columns : int
        number of columns in a projection
86    rows : int
        number of rows in a projection
88
      Returns
90    -------
      self
92    """
      self.IDsPerPixel_matrix = np.swapaxes(np.array([[len(self.matrix[:,i,j][self.
                                          matrix[:,i,j] != 0]) for i in range(0,rows
                                          )] for j in range(0,columns)]), 0,1)
```

Die Funktionen `makehist()` und `plot()` dienen zur graphischen Darstellung der dem untersuchten Merkmal zugeordneten Daten. `plot()` stellt die in `IDsPerPixel_matrix` abgelegte Projektion dar und `makehist()` erstellt eine Abbildung des Histogramms der Daten, welche ebenfalls durch die Funktion `plot()` ausgegeben wird.

```
  def makehist(self):
96    """reproduce the root-histogram
      """
98    nums = self.matrix.shape[0]*self.matrix.shape[1]*self.matrix.shape[2]
      custom_row_matrix = np.reshape(self.matrix, nums)
100   matrixwithoutzeros  = custom_row_matrix[custom_row_matrix!=0]
      self.hist = np.histogram(matrixwithoutzeros, 100)
102
  def plot(self, name, directory):
104   """plot the projection and the histogram

106   Parameters
```

```
    ----------
    name : str
      the name of the object
    directory : str
      the directory to save the plots
    """
    plt.imshow(self.IDsPerPixel_matrix, cmap = 'gray')
    plt.title(name)
    plt.savefig(directory + name + '.tif')
    plt.close()
    x = self.hist[1][1:len(self.hist[1])-1]
    y = self.hist[0][1:]
    plt.plot(x, y, '.')
    plt.ylim([0, int(max(y))])
    plt.title(name + '_histogram')
    plt.savefig(directory + name + '_histogram' + '.tif')
    plt.close()
```

Die Funktionen `getfft()`, `getfftshift()`, `getifft()` und `getifftshift()` dienen dazu, die Projektion (`IDsPerPixel_matrix`) in den Frequenzraum zu transformieren und die niedrigen Frequenzen außen oder in der Mitte (shift) des Bildes darstellen zu können.

```
  def getfft(self):
    """make a fourier transform of the IDsPerPixel matrix
    """
    nozeros = np.abs(np.fft.fft2(self.IDsPerPixel_matrix))
    nozeros[nozeros == 0] = 1
    self.IDsPerPixel_fft_matrix = np.log(nozeros)

  def getfftshift(self):
    """make fourier transform with the low frequencies in the center of the image
    """
    nozeros = np.abs(np.fft.fftshift(np.fft.fft2(self.IDsPerPixel_matrix)))
    nozeros[nozeros == 0] = 1
    self.IDsPerPixel_fft_matrix = np.log(nozeros)

  def getifft(self):
    """make an inverse fourier transform of the fourier transformed IDsPerPixel
                                        matrix
    """
    self.IDsPerPixel_fft_matrix = np.abs(np.fft.ifft2(np.exp(self.
                                        IDsPerPixel_fft_matrix)))

  def getifftshift(self):
    """make an inverse fourier transform of the shifted fourier transformed
                                        IDsPerPixel matrix
    """
    self.IDsPerPixel_fft_matrix = np.abs(np.fft.ifft2(np.fft.ifftshift(np.exp(self.
                                        IDsPerPixel_fft_matrix))))
```

Zur Ein- und Ausgabe dienen die Funktionen `savearrays()`, `loadarrays()` und `loadpicture()`. Während `savearrays()` und `loadarrays()` sämtliche erstellte Arrays (außer fouriertransformierte) speichern oder laden, lädt `loadpicture()` nur die Projektion.

```
  def savearrays(self, name, directory):
150   """save the numpy arrays in the directory

152   Parameters
      ----------
154   name : str
        the name of the object
156   directory : str
        the directory to save the arrays
158   """
      np.save(directory + name + '_projection' + '.npy', self.IDsPerPixel_matrix)
160   np.save(directory + name + '_matrix' + '.npy', self.matrix)
      np.save(directory + name + '_projectionvector' + '.npy', self.IDsPerPixel)
162   np.save(directory + name + '_matrixvector' + '.npy', self.vector)

164 def loadarrays(self, name, inputdir):
      """load the numpy arrays from the directory
166
      Parameters
168   ----------
      name : str
170     the name of the object
      inputdir : str
172     the directory to load the arrays from
      """
174   self.IDsPerPixel_matrix = np.load(inputdir + name + '_projection' + '.npy')
      self.matrix = np.load(inputdir + name + '_matrix' + '.npy')
176   self.IDsPerPixel = np.load(inputdir + name + '_projectionvector' + '.npy')
      self.vector = np.load(inputdir + name + '_matrixvector' + '.npy')
178
  def loadpicture(self, name, inputdir):
180   """load the projection-array from the directory

182   Parameters
      ----------
184   name : str
        the name of the object
186   inputdir : str
        the directory to load the projection-array from
188   """
      self.IDsPerPixel_matrix = np.load(inputdir + name + '_projection' + '.npy')
```

Zur Filterung der Daten eines Merkmals durch eine untere und eine obere Schranke der Werte dient die Funktion `filterminmax()`. Hierbei werden die Werte eines `data_matrices`-Objekts `toFilter` gefiltert und in `self` abgelegt.
Um die Werte zweier `data_matrices`-Objekte zusammenzufügen, kann ein Objekt `toJoin` in die `joinarrays()`-Funktion übergeben werden. `self` wird dann um die Werte in `toJoin` erweitert. Dies dient hauptsächtlich dazu, Simulationen unter gleichen Randbedinungen parallel berechnen zu können und sie im Nachhinein zusammenzufügen.

```
  def filterminmax(self, toFilter, minvalue, maxvalue):
192   """filter the data by a mininmal and a maximal value

194   Parameters
      ----------
196   toFilter : data_matrices object
        the object to filter and then write to self
198   minvalue : float
        the minimal value for the data in the toFilter object
200   maxvalue : float
        the maximal value for the data in the toFilter object
202   """
      self.vector = np.array(toFilter.vector)
204   self.vector[(self.vector <= minvalue)] = 0
      self.vector[(self.vector >= maxvalue)] = 0
206   self.reshape(self.columns, self.rows)
      self.makeprojection(self.columns, self.rows)
208
  def joinarrays(self, toJoin):
210   """join the array sets of two objects

212   Parameters
      ----------
214   toJoin : data_matrices object
        the object to join to self
216   """
      self.IDsPerPixel_matrix += toJoin.IDsPerPixel_matrix
218   self.matrix = np.append(self.matrix, toJoin.matrix, axis = 0)
      self.IDsPerPixel += toJoin.IDsPerPixel
220   self.vector = np.append(self.vector, toJoin.vector, axis = 0)
```

6.3 Zusammensetzen mehrerer Simulationen

Um auf mehreren Kernen eines Rechen-Clusters gleichzeitig simulieren zu können, werden mehrere Simulationen unter gleichen Bedingungen durchgeführt. Die Ergebnisse der Simulationen (ROOT Dateien) werden mit dem in Abschnitt 6.2.1 vorgestellten Skript in numpy Arrays abgelegt. Das im Folgenden präsentierte Skript **`adder.py`** dient dazu, mehrere Simulationen zusammenzufügen (vollständiges Skript im Anhang unter A.2.4). Im Vorhinein sind bereits die einzelnen (zeitlich unterscheidbaren) Schichten getrennt worden, sodass je Simulation mehrere Objektorientierungen aufgenommen und auch im Nachhinein zusammengefügt werden können.
In dem Skript werden zunächst die Bilddaten, wie Bildgröße und Pfade, definiert und daraus die Einlese- und Speicherverzeichnisse erstellt. Dann werden neue `data_matrices`-Objekte initialisert. `features` dient zur Speicherung der zusammengefügten Matrizen und `features_temp` lädt jeweils die neuen Arrays, die hinzugefügt werden sollen. So kumulieren sich die Daten aus diversen Messungen in einem `data_matrices`-Objekt und werden auch als solches gespeichert.

```
####### DEFINE INPUT/OUTPUT PARAMETERS #######
columns = 150
rows = 200
base_inputdir_1 = '/run/media/sj/SJ1/Bachelor/Data_sjung/simulations_4/
                                        thirdCT_400000000_120slices_'
base_inputdir_2 = '/arrays'
numinputdirs_1 = 50
numinputdirs_2 = 120
inputdirs = []
base_outputdir = '/run/media/sj/Seagate Expansion Drive/Data_sjung/cumulated_4/
                                        thirdCT_400000000_120slices_cumulated'
outputdirs = []

for i in range(0,numinputdirs_2):   #make input/output directories
  metainputdirs = []
  for j in range(0,numinputdirs_1):
    metainputdirs.append(base_inputdir_1 + str(j) + base_inputdir_2 + str(i) +'/')
  inputdirs.append(metainputdirs)
  outputdirs.append(base_outputdir + str(i) + '/')
  if not os.path.exists(outputdirs[-1]):
      os.mkdir(outputdirs[-1], 0755)

####### READ AND CUMULATE THE DATA #######
for k in range(0, numinputdirs_2):
  numfeatures = 1   #the number of features to cumulate
  features = [data_matrices(columns*rows, columns, rows) for _ in range(0,
                                        numfeatures)]   #empty data matrices
  features_temp = [data_matrices(columns*rows, columns, rows) for _ in range(0,
                                        numfeatures)]    #empty data matrices
  featurestrings = ['trackLength']

  for i in range(0, numinputdirs_1):
    print(inputdirs[k][i])
    for j in range(0, numfeatures):
      features_temp[j].loadarrays(featurestrings[j], inputdirs[k][i])   #load the
                                        single data arrays
      features[j].joinarrays(features_temp[j])    #cumulate by joining the new data
                                        arrays to the existing ones
    print('sum(features_temp[i][j].IDsPerPixel)', sum(features_temp[j].IDsPerPixel))
    print('sum(features[i][j].IDsPerPixel)', sum(features[j].IDsPerPixel))

  print(outputdirs[k])
  for j in range(0, numfeatures):
    features[j].savearrays(featurestrings[j], outputdirs[k])    #save the cumulated
                                        arrays
```

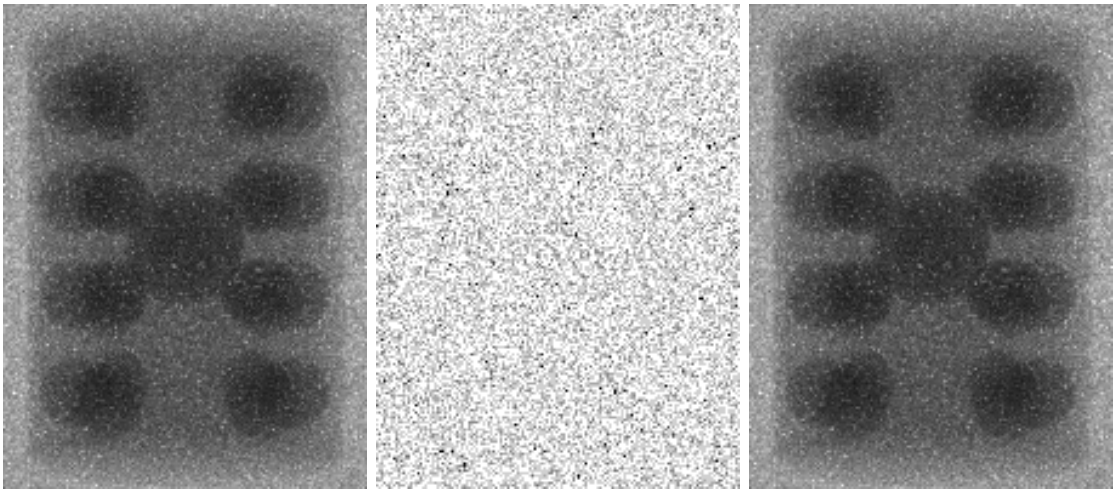
6.4 Filterung der Daten zur Erstellung von Trainingsdaten

Die Informationen des ROOT-Trees sollen verwendet werden, um aus den Simulationsdaten nahezu rauschfreie Projektionen zu erzeugen. Es werden verschiedene Ansätze getestet. Da die jeweils gefilterte Streustrahlung nur sehr geringe Grauwerte und eine geringe Dynamik umfasst,

werden die entstehenden Matrizen invertiert und mit der Potenz $\gamma = 0.1$ nicht-linear skaliert. Ohne Veränderung der Skalierung wären die Projektionen (insbesondere in gedruckter Form) nicht interpretierbar. Details zu dieser Skalierung sind in Abschnitt 8.1.1 nachzulesen.

6.4.1 Filterung durch physikalische Effekte

GATE loggt für jedes gezählte Photon die Anzahl der Compton- und Rayleigh-Streuprozesse im Detektor und im Phantom. Hier sind nur die Streuprozesse im Phantom relevant. Es lassen sich also gestreute Photonen von ungestreuten trennen. Lediglich Photonen, die indirekt aus dem Photo-Effekt eines anderen Photons gestreut werden lassen sich so nicht herausfiltern.
In Abbildung 6.1 sind verschieden gefilterte Projektionen eines Validierungsphantoms zu sehen. Abbildung 6.1a zeigt die ungefilterte Projektion. In 6.1b ist der im Phantom Compton-gestreute Anteil der Pixelwerte zu sehen. 6.1c zeigt selbiges für Rayleigh-Streuung und 6.1d nimmt beide Effekte zusammen. Nur aufgrund der nicht-linearen Skalierung sind hier Grauwerte zu erkennen. Es fällt somit auf, dass es nur wenige Pixel gibt, in denen viele gestreute Photonen zusammenkommen. Über den gesamten Detektor sind allerdings einzelne gestreute Photonen detektiert worden. Die Abbildungen 6.1e, 6.1f und 6.1g zeigen die, um die gestreuten Photonen reduzierten, Projektionen. Es ist nahezu keine Änderung zu erkennen. Die im Objekt Compton und Rayleigh gestreuten Photonen haben offensichtlich nur einen sehr kleinen Anteil an den gesamten detektierten Teilchen.



(a) (b) (c)

Abbildung 6.2 In Bild (a) ist die ungefilterte Projektion zu sehen. Bild (b) zeigt den Anteil der Pixelwerte, die durch Compton-gestreute Photonen entstanden sind (invertiert und mit $\gamma = 0.1$ potenziert). In Bild (c) sind die durch Compton-gestreute Photonen entstanden Zähler von der ursprünglichen Projektion abgezogen.

Dies entspricht nicht unbedingt der Erwartung. Zunächst ist angenommen worden, dass sämtliche Streuung im Phantom stattfindet. Da das umgebende Material (Luft) eine sehr geringe Dichte aufweist, also die Anzahl an Streuzentren deutlich geringer ist, als im Phantom, bleibt diese Annahme weiterhin berechtigt. Es ist zudem angenommen worden, dass die Compton-Streuung hier der dominierende Streuprozess ist (vgl. 2.2). Diese Annahme kann durch den photoelektrischen Effekt relativiert werden. Da in diesem Phantom das Anodenmaterial (Wolfram) vorkommt,

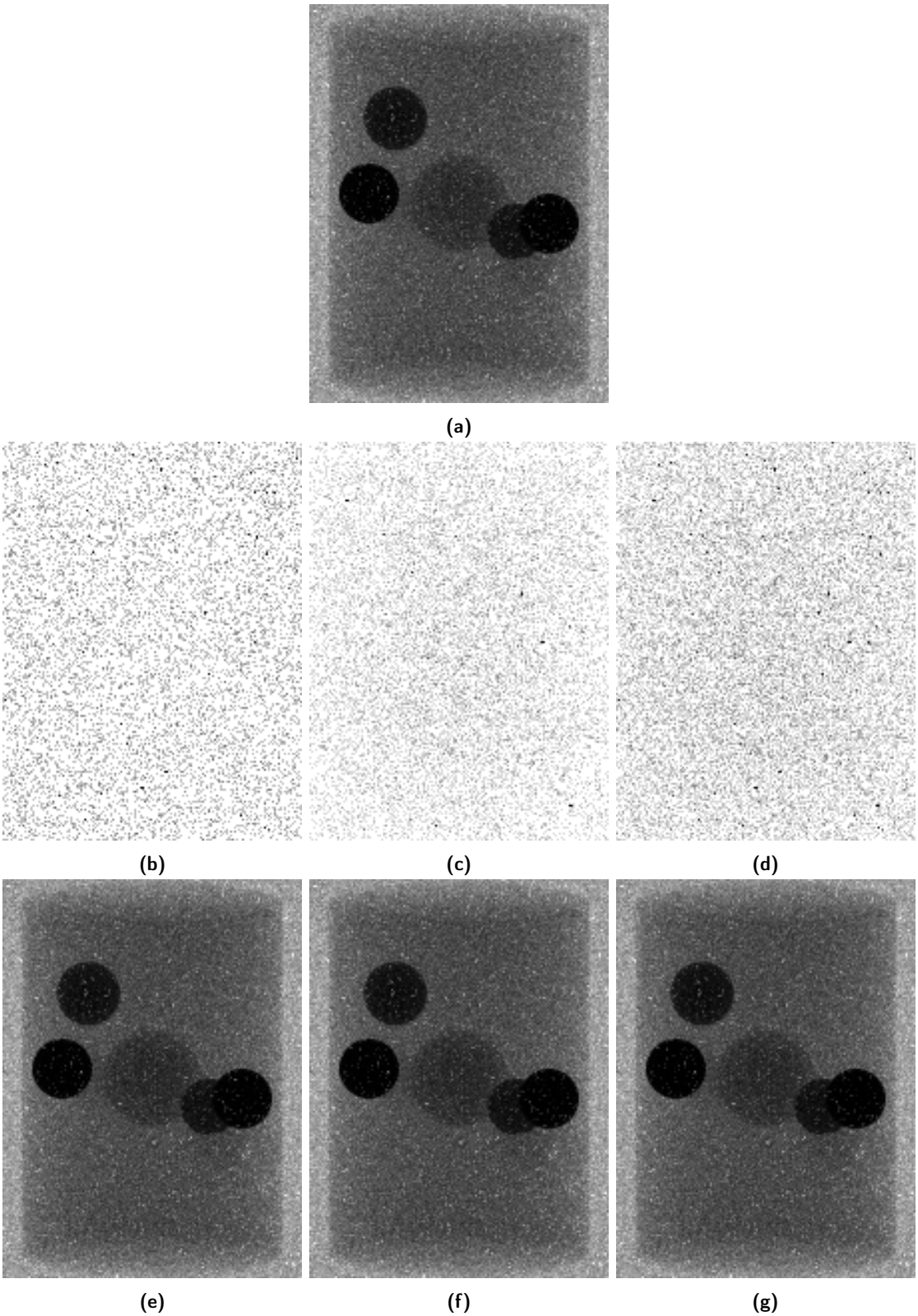
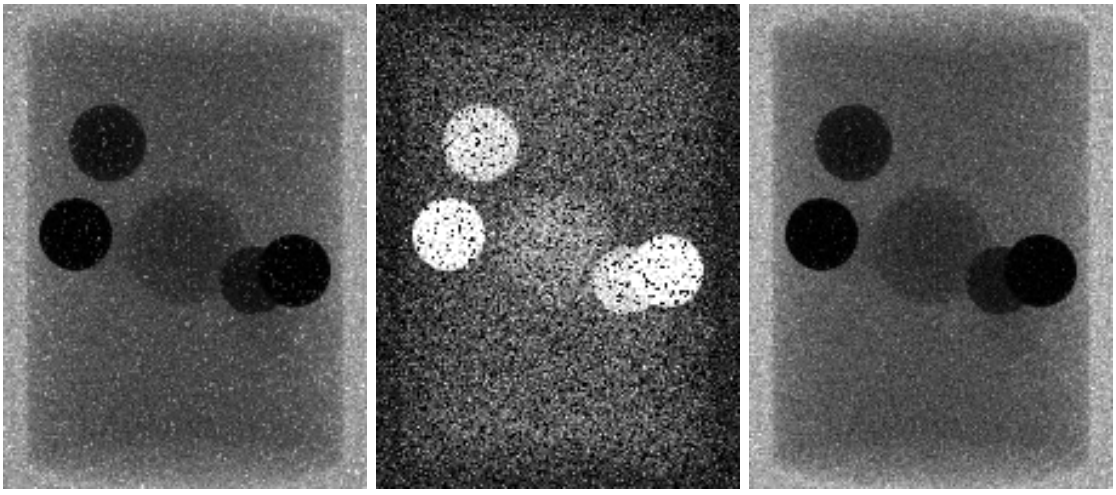


Abbildung 6.1 In Bild (a) ist die ungefilterte Projektion zu sehen. Bild (b) zeigt den Anteil der Pixelwerte, die durch Compton-gestreute Photonen entstanden sind. (c) zeigt selbiges für Rayleigh-gestreute Photonen ((b) und (c) sind invertiert und mit $\gamma = 0.1$ potenziert). In Bild (d) sind beide Anteile addiert. In Bild (e) sind die durch Compton-gestreute Photonen entstanden Zähler von der ursprünglichen Projektion abgezogen. (f) zeigt selbiges für Rayleigh-gestreute Photonen. In Bild (g) sind beide Anteile abgezogen.

steigt die Wahrscheinlichkeit für den Photoeffekt. Die Energien, die im Spektrum der Röntgenröhre (Abbildung 5.2) am häufigsten vorkommen, entsprechen hier genau den Anregungsenergien. Des Weiteren hat Wolfram eine hohe Ordnungszahl. Dies erhöht die Wahrscheinlichkeit für den Photoeffekt im verwendeten Energiebereich (vgl. Abbildung 2.2).
Um den Anteil des Photoeffekts an den gestreuten Photonen zu reduzieren, wird eine Simulation mit einem Ein-Material-Phantom durchgeführt (Datensatz III, siehe Abschnitt 5.4). Dieses Phantom besteht lediglich aus Aluminium, die resultierenden Projektionen sind in Abbildung 6.2 zu sehen. In diesem Fall ist lediglich die Compton-Streuung betrachtet.
Es ergeben sich bei der Betrachtung keine Unterschiede zu der Filterung des Phantoms, welches das Targetmaterial beeinhaltet.
Offensichtlich ist in den Projektionen auch nach der Reduktion um Compton- und Rayleigh-gestreute Photonen noch Rauschen vorhanden. Es wird nicht quantitativ untersucht, warum diese Prozesse hier einen Anteil dieser Größenordnung zur Projektion beitragen. Es gilt zusätzlich, den Photoeffekt als indirekten Streuprozess hinzuzuziehen. Eine quantitative Untersuchung der, aus dem Photoeffekt resultierenden Photonen, wird nicht durchgeführt.

6.4.2 Filterung durch Energieverteilung

Bei Compton-Streuung geben die Photonen abhängig vom Streuwinkel einen Teil ihrer Energie ab. Infolgedessen umfasst das Spektrum der gestreuten Strahlung geringere Energien als das ursprüngliche Röntgenspektrum. Je höher die Energie des Photons, desto weniger stark wird es gestreut (vgl. Abbildung 2.1). Die Wahrscheinlichkeit für Compton-gestreute Photonen ist also bei niedrigen Energien deutlich erhöht. Insofern bietet sich die Möglichkeit an, eine Projektion lediglich aus Pixelwerten zusammenzustellen, welche eine gewisse Energieschwelle überschreiten.



(a) (b) (c)

Abbildung 6.3 In Bild (a) ist die ungefilterte Projektion eines verschiedene Materialien umfassenden Phantoms zu sehen. In Bild (b) sind die durch Photonen außerhalb des Energiefensters ([5keV, 600keV]) entstanden Zähler zu sehen (invertiert und mit $\gamma = 0.1$ potenziert). (c) zeigt die Projektion, die lediglich aus Photonen innerhalb es Energiefensters besteht.

In Abbildung 6.3 ist eine derartige Filterung zu sehen. Es wird zunächst eines der Validierungsphantome betrachtet, welches aus verschiedenen Matererialien zusammengesetzt ist (vgl.

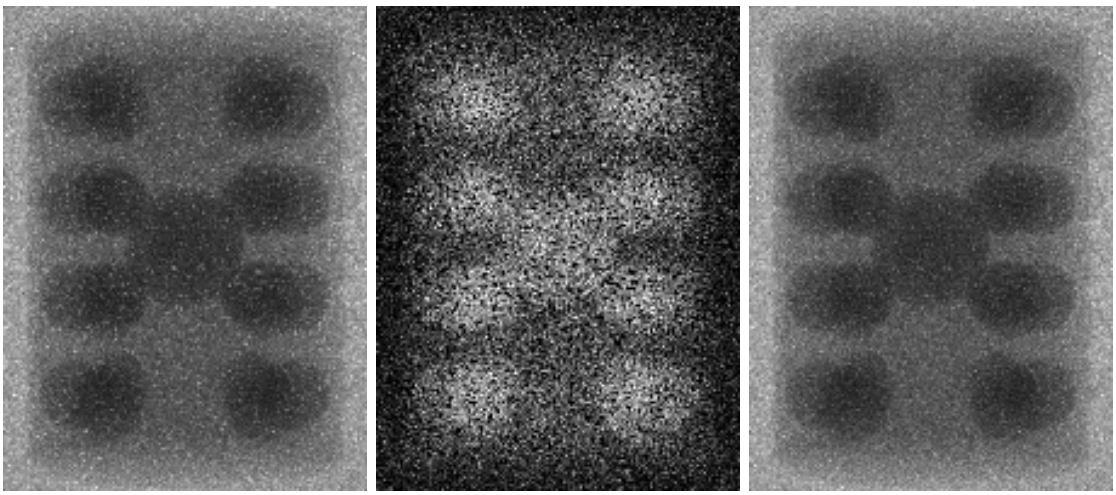
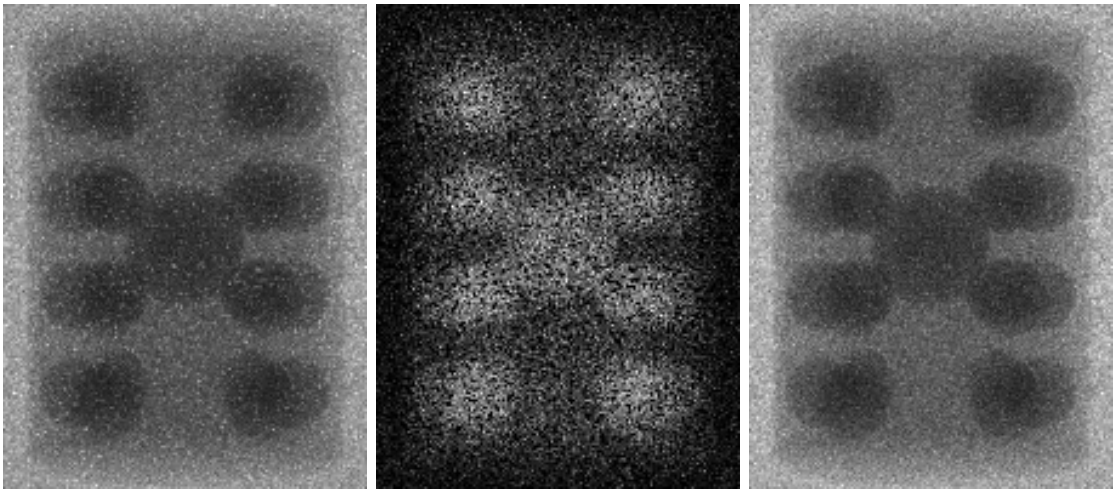


Abbildung 6.4 In Bild (a) ist die ungefilterte Projektion eines lediglich Aluminium beinhaltenden Phantoms zu sehen. In Bild (b) sind die durch Photonen außerhalb des Energiefensters ([3keV, 600keV]) entstanden Zähler zu sehen (invertiert und mit $\gamma = 0.1$ potenziert). (c) zeigt die Projektion, die lediglich aus Photonen innerhalb des Energiefensters besteht.

Abschnitt 5.4). 6.3a zeigt die ungefilterte Projektionsmatrix. 6.3b und 6.3c teilen die Projektion in die Zähler auf, welche unterhalb und oberhalb einer Energieschwelle von 5keV entstanden sind. Es ist zu erkennen, dass unterhalb von 5keV hauptsächlich Bildrauschen bzw. Streustrahlung enthalten ist, da erstens nur wenig Objektinformationen zu erkennen sind und zweitens bei Vergleich des Originals und der Projektion mit Energien >5keV das Rauschen deutlich reduziert ist.

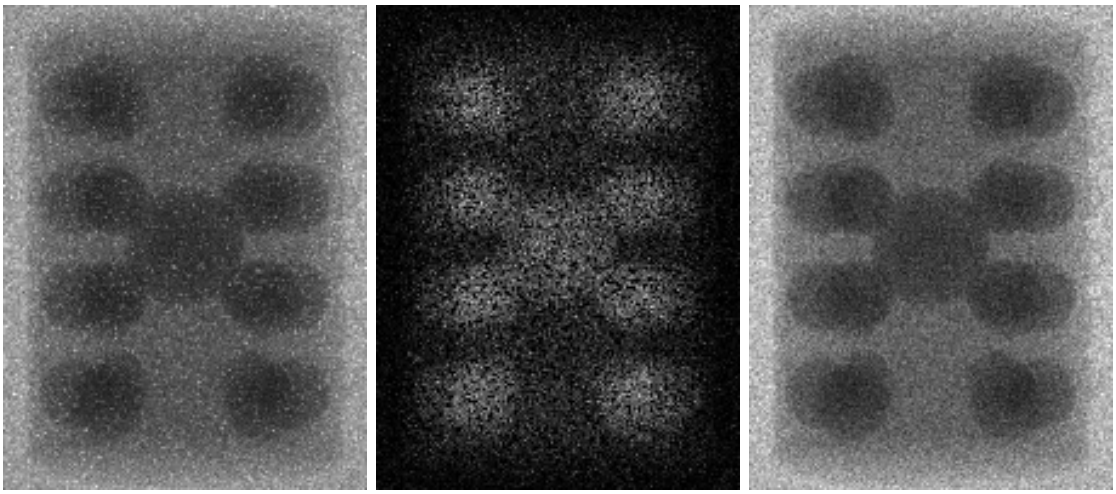
Eine solche Energieschwelle schließt ebenfalls die ungestreute Strahlung niedriger Energie aus. Diese kann allerdings gerade für Materialien geringer Ordnungszahl entscheidende Informationen enthalten, da die Strahlung höherer Energie hier nahezu absorptionslos durchdringt. Deshalb wird als nächstes ein Objekt untersucht, was lediglich aus einem Material (Alumnium, Datensatz III) besteht. Aluminium hat eine Ordnungszahl von 13. Insofern wird hier immer noch die Streuung hauptsächlich niedrig energetischer Strahlung erwartet. Da lediglich ein Material vorhanden ist, kann der Einfluss der Schwelle genauer untersucht werden.

In Abbildung 6.4 ist eine Filterung durch die Energieschranke 3keV, die der niedrigsten Energie des erzeugten Spektrums entspricht, dargestellt. Abbildung 6.5 zeigt eine Filterung für Energien größer bzw. kleiner als 5keV und 6.6 setzt die Schranke auf 10keV. Bei Vergleich der Filterungen mit 3keV und 5keV zeigen die Matrizen niedriger Energie (6.4b und 6.5b) mehr Zähler, wenn die Schranke bei 5keV liegt, aber es ist nicht viel mehr Objektinformation zu erkennen. Die Matrizen hoher Energie (6.4c und 6.5c) unterscheiden sich nicht sehr. Beim Vergleich von 5keV und 10keV fällt auf, dass in der Matrix höherer Energie (Schranke 10keV, 6.6b) das Objekt deutlicher und genauer zu erkennen ist. Die Matrix hoher Energie (6.6c) wirkt rauschfreier als die Matrizen der anderen Energieschranken. Dennoch darf der größere Informationsverlust hierbei nicht vernachlässigt werden.



(a) (b) (c)

Abbildung 6.5 In Bild (a) ist die ungefilterte Projektion eines lediglich Aluminium beinhaltenden Phantoms zu sehen. In Bild (b) sind die durch Photonen außerhalb des Energiefensters ([5keV, 600keV]) entstanden Zähler zu sehen (invertiert und mit $\gamma = 0.1$ potenziert). (c) zeigt die Projektion, die lediglich aus Photonen innerhalb des Energiefensters besteht.



(a) (b) (c)

Abbildung 6.6 In Bild (a) ist die ungefilterte Projektion eines lediglich Aluminium beinhaltenden Phantoms zu sehen. In Bild (b) sind die durch Photonen außerhalb des Energiefensters ([10keV, 600keV]) entstanden Zähler zu sehen (invertiert und mit $\gamma = 0.1$ potenziert). (c) zeigt die Projektion, die lediglich aus Photonen innerhalb des Energiefensters besteht.

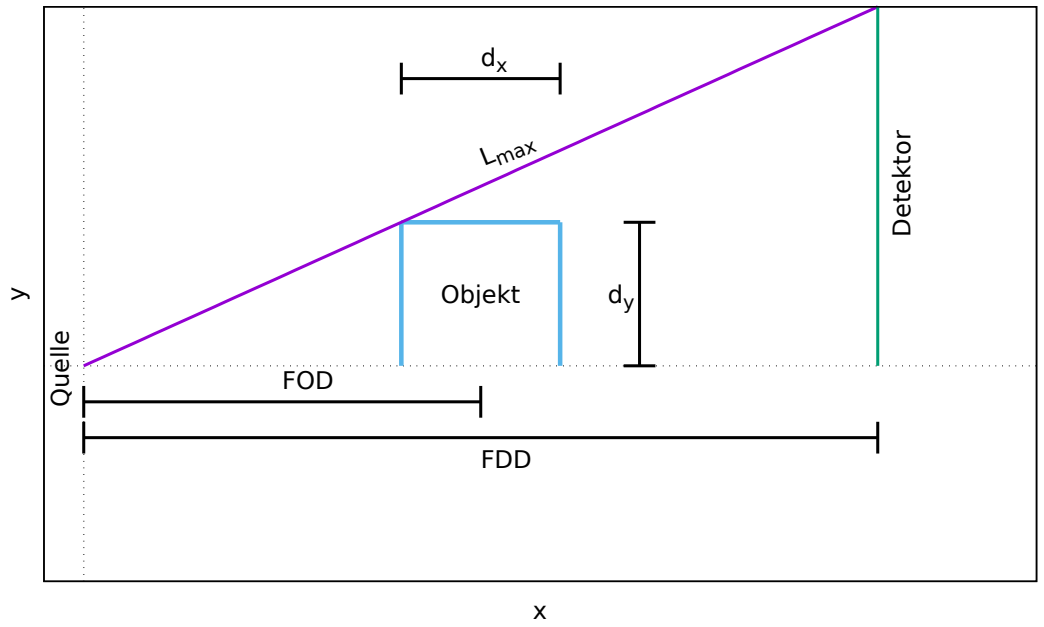


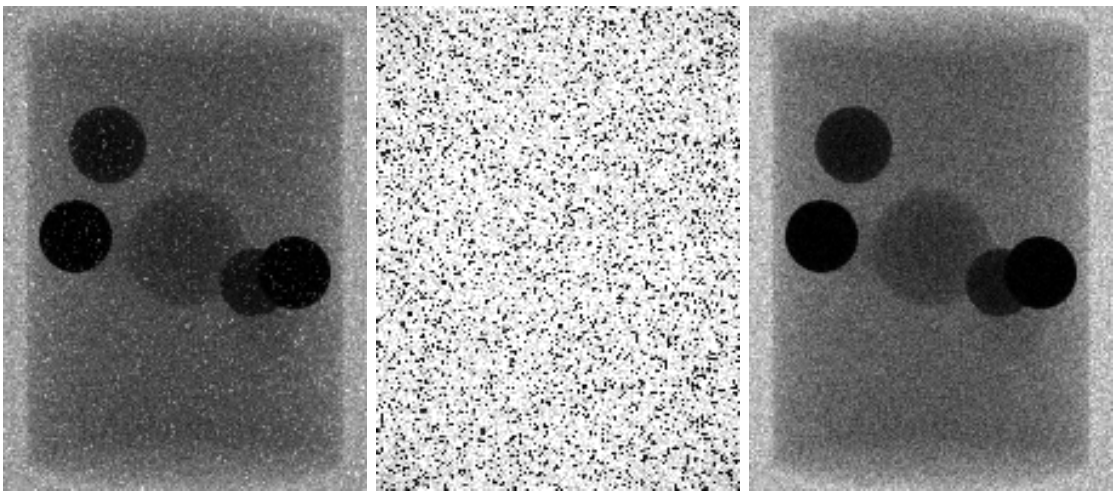
Abbildung 6.7 Skizze zur Maßdefinition der Röntgenanlage. Die skizzierten Größen dienen zur Filterung anhand der zurückgelegten Weglänge der Photonen.

6.4.3 Filterung durch TrackLength

Ein weiterer Ansatz zur Erzeugung rauschfreier Daten ist eine geometrische Filterung. GATE gibt zu jedem gezählten Photon den Parameter **`trackLength`** aus. Dieser Wert umfasst die zurückgelegte Distanz eines Photons, bevor es auf den Detektor trifft. Wird ein Photon gestreut, so weicht es von der direkten Gerade zwischen Quelle und Detektor ab. Genau das ist der Grund, warum solche Photonen keinen Rückschluss auf das Objekt zulassen. Es kann jedem Pixel eine kürzeste Distanz zur Röntgenquelle zugeordnet werden. Übersteigt der **`trackLength`**-Parameter diese Distanz hat ein Streuprozess stattgefunden. Bei indirekten Streuprozessen können auch kürzere Distanzen möglich sein. Es wird nicht weiter nachgeprüft, wie GATE diesen Parameter für sekundär erzeugte Photonen bestimmt. In erster Näherung reicht es, eine obere und eine untere Schranke für die Streustrahlung in allen Pixeln zu definieren. Es wird die maximal mögliche Distanz $L_{\max}$ bestimmt, die ein Teilchen zwischen der Röntgenquelle und dem Detektor zurücklegen kann, unter der Bedingung, dass es dabei das Phantom passiert. Als $L_{\min}$ wird die minimale Distanz zwischen Quelle und Detektor definiert. Die Gleichungen (6.1) und (6.2) geben diese Bedingungen an. Hierbei ist FDD (Fokus-Detektor-Distanz) die Distanz zwischen der Röntgenquelle und dem Detektor. FOD (Fokus-Objekt-Distanz) ist die Distanz zwischen Röntgenquelle und Phantom. d_x ist die Ausdehnung des Phantoms in der Detektor-Quelle-Achse und d_y orthogonal dazu die maximale Ausdehnung des Phantoms. Es wird jeweils ein Faktor $\lambda_{\min} < 1$ und $\lambda_{\max} > 1$ multipliziert, um die gesamte Detektorfläche auszunutzen. In Abbbildung 6.7 ist eine Skizze zu sehen, die die geometrischen Zusammenhänge erläutert.

$$L_{\max} = \lambda_{\max} \sqrt{FDD^2 + \left(\frac{FDD}{(FOD - \frac{d_x}{2})} \frac{d_y}{2} \right)^2} \tag{6.1}$$

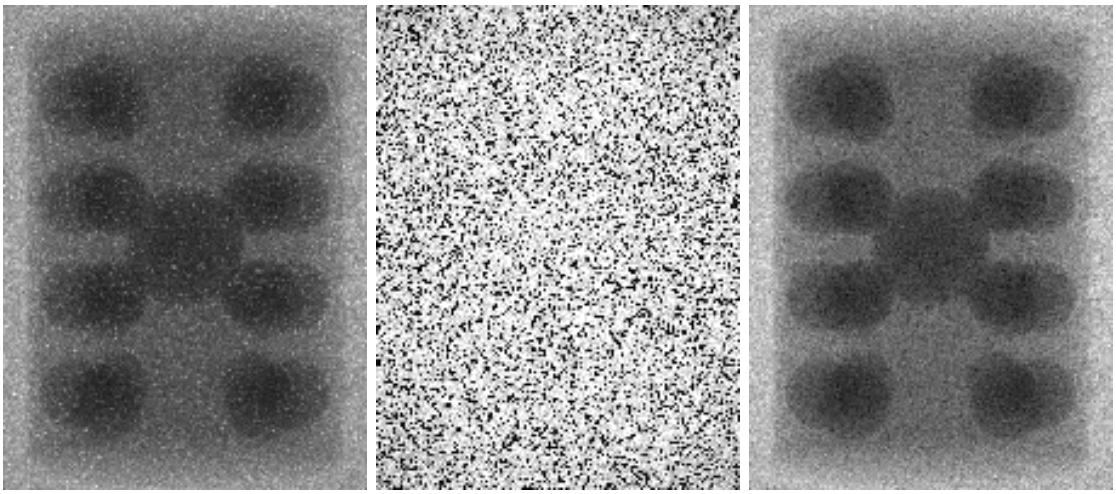
$$L_{\min} = \lambda_{\min} FDD \tag{6.2}$$



(a) (b) (c)

Abbildung 6.8 In Bild (a) ist die ungefilterte Projektion eines Validierungsphantoms zu sehen. In Bild (b) sind die durch Photonen, deren zurückgelegte Weglängen außerhalb des Rahmens waren, entstandenen Zähler zu sehen (invertiert und mit $\gamma = 0.1$ potenziert). (c) zeigt die Projektion, die lediglich aus Photonen innerhalb der geometrischen Schranken besteht.

Abbildung 6.8 zeigt die Anwendung der beschriebenen Filterung auf die Aufnahme eines Validierungsphantoms. Abbildung 6.8a zeigt die ungefilterte Projektion, 6.8b die durch diese Filterung definierte Streustrahlung und 6.8c das resultierende gefilterte Bild. Selbiges ist in Abbildung 6.9 für das, lediglich aus Aluminium bestehende, Phantom aus Datensatz III zu sehen.



(a) (b) (c)

Abbildung 6.9 In Bild (a) ist die ungefilterte Projektion des Phantoms III zu sehen. In Bild (b) sind die durch Photonen, deren zurückgelegte Weglängen außerhalb des Rahmens waren, entstandenen Zähler zu sehen (invertiert und mit $\gamma = 0.1$ potenziert). (c) zeigt die Projektion, die lediglich aus Photonen innerhalb der geometrischen Schranken besteht.

Bei beiden Simulationen ist in den gefilterten Projektionen die Streustrahlung deutlich reduziert und die herausgefilterte Strahlung lässt keinerlei Objektstrukturen erkennen. Im Vergleich zu den Filterungen durch die Rayleigh- und Compton-Streuung (vgl. 6.4.1) und die Energieverteilung (vgl. 6.4.2) wird hier am wenigsten Objektinformation herausgefiltert. In der Aufnahme des Validierungsphantoms ist in der obersten Kugel zu erkennen, dass hier der geringste Anteil

Streustrahlung übrig bleibt.
Als Verbesserung ist vorstellbar, die minimale und maximale Länge nicht durch das Objekt, sondern durch den Detektor vorzugeben. Hierdurch würde eine größere Allgemeinheit erreicht. Optimal wäre es, für jedes Pixel eine individuelle Diskriminierung vorzunehmen.

Die Filterung durch den Parameter **`trackLength`** liefert die besten Resultate und wird zur Erzeugung von Trainingsdaten verwendet. Zur Filterung wird ein zusätzliches Skript **`filter_by_trackLength.py`** geschrieben, welches im Anhang unter A.2.5 einsehbar ist.

7 Erstellung eines neuronalen Netzes

Im Folgenden werden die mathematischen Methoden sowie die konkrete Implementierung des neuronalen Netzes und der Vorverarbeitung der Daten dargestellt.

7.1 Konzept der pixelweisen Verarbeitung

Das aufgebaute neuronale Netz basiert auf einem pixelweisen Ansatz. Da für jedes Pixel entschieden werden muss, welcher Anteil seines Wertes aus Streustrahlung resultiert, wird auch jedes Pixel einzeln verarbeitet. Zur Erkennung von Streustrahlung werden Informationen der umgebenden Pixel benötigt. Diese Informationen werden jedem Pixel als Merkmale übergeben. Denkbare Merkmale sind beispielsweise die Werte der umliegenden Pixel.
Um diese Informationen effizient zu erfassen, werden - ähnlich wie bei Convolutional neural networks - Faltungskerne und Pooling-Verfahren zur Merkmalerstellung angewendet. Bei den Faltungskernen wird darauf geachtet, dass sie die Größe des Bildes nicht beinflussen, damit jedem Pixel ein Wert als Merkmal zugeordnet wird. Die Pooling-Verfahren dienen dazu, jedem Pixel als Merkmal einen Wert, der aus einer Funktion auf den Umgebungspixeln resultiert, zu geben. Beispielsweise das Maximum oder die Standardabweichung in der Dreiernachbarschaft.
Die so entstehenden Merkmalsmatrizen haben die gleiche Dimensionalität wie die Ausgangsbilder und bilden mit diesen einen Bildstapel. Ein Eingabedatensatz ist somit ein Bildpixel mit den Werten der Merkmalsmatrizen an den gleichen Koordinaten.

7.2 Vorverarbeitung der Projektionen zur Erstellung der Merkmale

Das erstellte Programm teilt sich in mehrere Klassen und ein Hauptprogramm auf. In dem Hauptprogramm findet die grundlegende Dateneingabe und -ausgabe statt. Hierzu sind zwei Funktionen (`load_tracklength_filtered()` und `min_max_scale()`, siehe 7.2.1), definiert. Des Weiteren werden sämtliche erforderliche Parameter definiert. Diese sind zunächst Informationen über Ein- und Ausgabepfade, Bildgröße und Datentyp sowie für die Merkmalerstellung die Filter-Kerne und als Parameter für das neuronale Netz die Anzahl der Durchläufe und sämtliche Hyperparameter, welche die Anzahl und Größe der Schichten, die Lernrate und die zu verarbeitende Stapelgröße umfassen.
Die in Abschnitt 6.2.2 bereits eingeführte Klasse `data_matrices` dient hier zum Einlesen der Simulationsdaten und zur Weitergabe an die Merkmalerstellung.
Zur Merkmalerstellung wird die Klasse `convolve_class` erstellt. Diese Klasse umfasst alle weiterführenden Funktionen zur Datenvorverarbeitung, wie Skalierung und die Merkmalerstellung. In Abschnitt 7.2.2 wird die Klasse eingeführt.
Das eigentliche neuronale Netz ist in der Klasse `NetWork` implementiert. Hier findet die Optimierung der Gewichte statt. Die Variation der Hyperparameter findet dagegen im Hauptprogramm statt. `NetWork` ist in Abschnitt 7.3 dokumentiert.

7.2.1 Einlesen der Daten

Zunächst werden die ungefilterten (trackLength) sowie gefilterten Datensätze (trackLengthwindow) in der Funktion load_trackLength_filtered() eingelesen. Diese Funktion ist Teil des Hauptprogramms des neuronalen Netzes, welches im Anhang unter A.2.6 zu finden ist. Die Funktion benötigt als Eingabgeparameter die Projektionsdimensionen (columns, rows) sowie den Pfad zu den Dateien (base_inputdir) und die Anzahl der einzulesenden Dateien (numinputdirs). Die Datensätze werden zunächst als data_matrices-Objekte initialisiert (Erläuterung der Klasse, siehe Abschnitt 6.2.2) und dann, mithilfe einer for-Schleife über die Pfade, eingelesen.
Die Parameter num_trainsets und num_testsets dienen der Trennung der Trainings und Testdaten. selected_data_range dient zur Skalierung der Ausgabebilder mithilfe der min_max_scale()- oder der min_custommax_scale()-Funktion. In diesen Funktionen wird das Ausgabebild entweder bezüglich ihres maximalen Wertes oder bezüglich eines Wertes custommax auf den gesamten, durch den Datentyp gewählten, Wertebereich skaliert.

```
def load_trackLength_filtered(columns, rows, numinputdirs, base_inputdir):
  """load the trackLength and trackLenthwindow projections, meaning the unfiltered
                                        and filtered data

  Parameters
  ----------
  columns : int
    number of columns
  rows : int
    number of rows
  numinputdirs : int
    number of images to load
  base_inputdir : str
    path to the images

  Returns
  -------
  trackLength : list, shape = [numinputdirs]
    trackLength[i] : data_matrices
    each element includes the projection
  trackLengthwindow : list, shape = [numinputdirs]
    trackLengthwindow[i] : data_matrices
    each element includes the projection, windowed by the trackLength
  """
  trackLength = [data_matrices(columns*rows, columns, rows) for _ in range(0,
                                        numinputdirs)]   #initialize lists
  trackLengthwindow = [data_matrices(columns*rows, columns, rows) for _ in range(0,
                                        numinputdirs)]
  for j in range(0, numinputdirs):
    inputdir = base_inputdir + str(j)+'/'   #set inputdirectory
    print(inputdir)
    trackLength[j].loadpicture('trackLength', inputdir)   #load the data
    trackLengthwindow[j].loadpicture('trackLengthwindow', inputdir)   #load the data
  return trackLength, trackLengthwindow

```

```
def min_max_scale(image, data_range = 'uint16'):
  """scale the image with its max value to the maximum of the data type

  Parameters
  ----------
  image : array, shape = [columns, rows]
    the image to scale
  data_range : string
    the name of the data type

  Returns
  -------
  image.astype(data_range) : array, shape = [columns, rows]
    the scaled image
  """
  if np.max(image) > 0:
    image*=(np.iinfo(data_range).max/np.max(image))
  return image.astype(data_range)

def min_custommax_scale(image, custommax, data_range = 'uint16'):
  """scale the image with to the the data type considering the given maximum

  Parameters
  ----------
  image : array, shape = [columns, rows]
    the image to scale
  custommax : float
    the Value that should be assigned to maximal available value of the data type
  data_range : string
    the name of the data type

  Returns
  -------
  image.astype(data_range) : array, shape = [columns, rows]
    the scaled image
  """
  if custommax > 0:
    image*=(np.iinfo(data_range).max/custommax)
  return image.astype(data_range)
```

7.2.2 Klasse zur Merkmalerstellung

Die Klasse `convolveWork` wird mit dem ungefilterten (`X`) und dem gefilterten Bild (`y`) initialisiert (gesamte Klasse, siehe A.2.7). Initialisiert wird zusätzlich eine leere Liste `features`, welche später die Merkmalsmatrizen zwischenspeichert. Die Parameter `X_rescale` und `y_rescale` sind Faktoren zur späteren Skalierung, die zunächst mit 1 initialisiert werden.

```
class convolveWork(object):

  def __init__(self, X, y):
    """

    Parameters
    ----------
    X : array, shape = [rows, columns]
      input image
    y : array, shape = [rows, columns]
      desired output image
    """
    self.X = X
    self.y = y
    self.features = []
    self.X_rescale = 1
    self.y_rescale = 1
```

Abwandlung sogenannter *Pooling-Layers* zur Merkmalerzeugung

Das sogenannte Pooling wendet eine Funktion auf ein Pixel und seine Umgebungspixel an. Die Umgebung umfasst hier einen Bereich der Größe `kernelsizex`×`kernelsizey` um ein Pixel und ordnet dem mittleren Pixel das Ergebnis der Funktion zu.
In dieser Untersuchung werden lediglich relativ simple Ansätze für das Pooling verwendet. Das Konzept ist beliebig erweiterbar. Hier werden unter dem Oberbegriff Pooling eine Maximum-, Minimum-, Mittelwert und Standardabweichungsfunktion verwendet.

```
  def maxpool(self, kernelsizex = 3, kernelsizey = 3):
    """give pixel the maximum value of area kernelsizex*kernelsizey around it

    Parameters
    ----------
    X : array, shape = [rows, columns]
      input image
    kernelsizex : int
      rows the kernel is regarding
    kernelsizey : int
      columns the kernel is regarding

    Returns
    -------
    pooled : array, shape = [rows, columns]
      maximum pooled matrix
    """
    pooled = filters.maximum_filter(self.X, size = (kernelsizex, kernelsizey), mode
                                    = 'constant')
    return pooled
```

Die `maxpool`-Funktion nutzt die `scipy.ndimage.filters.maximum_filter`-Funktion, welche jedem Pixel das Maximum der `kernelsizex`×`kernelsizey` Umgebung zuordnet. Die Wahl `mode = 'constant'` erweitert die Ränder des Bildes um Nullen, damit auch hier eindeutig das

Maximum berechnet werden kann.
Selbiges Verfahren wird zur Bestimmung der Minimums- und Mittelwert-Matrizen verwendet. Hier werden die Funktionen `minimum_filter` und `uniform_filter` verwendet (siehe Anhang A.2.7).
Zur effizienten Berechnung der Standardabweichung wird zweifach der `uniform_filter` angewendet, sowohl auf die Werte als auch auf die Quadrate der Werte. Hiermit kann die Standardabweichung berechnet werden. Es kann hier allerdings zu Problemen im Aufruf der Wurzelfunktion kommen. Insofern werden zu `X` kleine Zufallszahlen hinzugefügt, um das Wurzelziehen einer negativen Zahl zu vermeiden [29].

```
  def stdpool(self, kernelsizex = 3, kernelsizey = 3):
86  #SOURCE: https://nickc1.github.io/python,/matlab/2016/05/17/Standard-Deviation-(
                                        Filters)-in-Matlab-and-Python.html
    """give pixel the standard-deviation value of area kernelsizex*kernelsizey
                                        around it
88
    Parameters
90  ----------
    X : array, shape = [rows, columns]
92    input image
    kernelsizex : int
94    rows the kernel is regarding
    kernelsizey : int
96    columns the kernel is regarding

98  Returns
    -------
100 pooled : array, shape = [rows, columns]
      standard-deviation pooled matrix
102 """
    X = self.X + np.random.rand(self.X.shape[0], self.X.shape[1])*1e-5    #to not
                                        get nans in the np.sqrt() call
104 meanX = filters.uniform_filter(X, size = (kernelsizex, kernelsizey), mode = '
                                        constant')   #calculate the mean
    meanXsq = filters.uniform_filter(X*X, size = (kernelsizex, kernelsizey), mode =
                                        'constant')   #calculate the mean of the
                                        squares
106 return np.sqrt(meanXsq-meanX*meanX)   #calculate the standard deviaton
```

Abwandlung sogenannter *Convolutional-Layer* zur Merkmalerzeugung

Zur weiteren Merkmalerstellung werden elementweise Bildfaltungen mit kleinen Faltungskernen durchgeführt. Eine zweidmensionale diskrete Faltung des Bildes $I(x, y)$ mit der Matrix $k \in \mathbb{R}^{d_x \times d_y}$ wird durch

$$I * k = \sum_{i=-d_x}^{d_x} \sum_{i=-d_y}^{d_y} I(x-i, y-j)k(i, j) \tag{7.1}$$

ausgedrückt [30, S. 43]. Eine Bildfaltung gewichtet immer die Umgebung eines Pixels mit den im Faltungskern enthaltenen Werten. Es können hierdurch Strukturen wie beispielsweise Kanten

hervorgehoben werden. Einige Beispiele für solche Matrizen sind

$$k_{11} = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix}, \qquad k_{12} = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}, \tag{7.2}$$

$$k_{21} = \begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}, \quad k_{22} = \begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix}. \tag{7.3}$$

Da Kanten unterschiedliche Richtungen haben können, werden je zwei Filtermatrizen aufgestellt. Ob der Wechsel von großen zu kleinen oder kleinen zu großen Werten stattfindet, wird durch das Vorzeichen unterschieden. Die k_1 Matrizen stellen den Sobel-Operator dar, welcher die erste Ableitung in der jeweiligen Richtung darstellt. Der Prewitt-Operator ist in den k_2-Matrizen zu sehen. Dieser ist dem Sobel-Operator sehr ähnlich und unterscheidet sich offensichtlich nur durch eine identische Gewichtung aller einfließenden Werte [30, S. 47].

$$k_3 = \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}, \qquad k_4 = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}, \tag{7.4}$$

$$k_5 = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}, \qquad k_6 = \begin{pmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{pmatrix}. \tag{7.5}$$

Weitere Kantenfilter sind in k_3 und k_5 dargestellt. k_6 stellt als sogenannte Laplace-Filter die zweite Ableitung in der Bildebene dar. k_4 ist eine Abwandlung des Laplace-Filters, welcher auch diagonale Kanten erkennt.

```
108   def conv_w_kernel(self, kernel):
        """convolvle the image X with the kernel
110
        Parameters
112     ----------
        X : array, shape = [rows, columns]
114       input image
        kernel: array, shape = [kernelsizex, kernelsizey]
116       the kernel to convolve with

118     Returns
        -------
120     convolved : array, shape = [rows, columns]
          convolvedd image
122     """
        convolved = signal.convolve2d(self.X, kernel, mode = 'same')
124     return convolved
```

Die Faltungsoperation findet in der Funktion `conv_w_kernel()` statt. Diese Funktion faltet das Bild in der Variable `X` mit dem Kern `kernel`. Hierzu wird die Funktion `scipy.signal.convolve2d` verwendet. Der Modus `mode = 'same'` bedeutet hier, dass das Ausgabebild die gleiche Größe wie das Eingabebild hat. Der Bildrand wird hierfür um Nullen ergänzt.

Weitere Datenvorverarbeitung

In der Klasse convolveWork stehen weitere Funktionen zur Verfügung, um die Daten vorzuverarbeiten. In der Funktion vectorize werden sämtliche Bild-, als auch Merkmalsmatrizen vektorisiert und die Bildmatrizen können skaliert werden. Dies ist für eine allgemeine Anwendbarkeit sinnvoll. Hierzu dient die Funktion _scale. Standardmäßig werden die Bilder auf ein Intervall [0,1] skaliert. Sie können allerdings auch durch Übergabe der Parameter X_cuscale und y_cuscale mit bereits definierten Faktoren skaliert werden.

```
126   def vectorize(self, do_scale = True, set_scale = False, custom_scale = False,
                                          X_cuscale = 1, y_cuscale = 1):
        """vectorize image and features for better handling
128
        Parameters
130     ----------
        do_scale : Bool
132       scale the pixel values between 0 and 1 or not

134     Returns
        -------
136     X_vectorized : array, shape = [n_pixels, n_features+1]
          with n_pixels = rows*columns
138     y_vectorized : array, shape = [n_pixels]
        """
140     self.X_vectorized = np.empty((self.X.shape[0]*self.X.shape[1], len(self.features
                                          )+1))
        self.X_vectorized[:,0] = self.X.reshape((self.X.shape[0]*self.X.shape[1]))
142     i = 1
        for feature in self.features:
144       self.X_vectorized[:,i] = feature.reshape(((self.X.shape[0]*self.X.shape[1])))
                                            #will raise error, if feature dimension
                                          mismatches picture dimension
          i+=1
146     self.y_vectorized = self.y.reshape(self.y.shape[0]*self.y.shape[1])
        if do_scale:
148       if set_scale:
            self.X_rescale = np.max(self.X_vectorized, axis = 0)
150         self.y_rescale = np.max(self.y_vectorized)
          if custom_scale:
152         self.X_rescale = X_cuscale
            self.y_rescale = y_cuscale
154       self._scale()
        return self.X_vectorized, self.y_vectorized, self.X_rescale, self.y_rescale
156
      def _scale(self):
158     """scale the data to the interval [0,1]
        """
160     self.y_vectorized *= 1/self.y_rescale
        self.X_vectorized *= 1/self.X_rescale
162     return self
```

Die Funktion rescale_and_reshape() dient dazu, die Ein- und Ausgabematrizen sowie die optimierte Bildmatrix zurückzuskalieren. Da die optimierte Matrix negative Werte enthalten

kann, werden diese vorher auf den Wert 0 gesetzt.

```
164   def rescale_and_reshape(self, X, y, optimized, rows, columns):
        """ rescale the data to the original absolute bandwidth
166
        Parameters
168     ----------
        X : array, shape = [n_pixels]
170       the input image, the first dimension of the feature matrix
        y : array, shape = [n_pixels]
172       the desired image
        optimized : array, shaoe = [n_pixels]
174       the output image of the NetWork
        max_scale : Bool
176       scale each image to its own maximum
        data_range : str
178       Which datatype is selected for the max-scaling

180     Returns
        -------
182     X_out : array, shape = [rows, columns]
          the input image, the first dimension of the feature matrix
184     y_out : array, shape = [rows, columns]
          the desired image
186     optimized_out : array, shaoe = [rows, columns]
          the output image of the NetWork
188
        """
190     X_out = (self.X_rescale[0]*X).reshape((rows, columns))
        y_out = (self.y_rescale*y).reshape((rows, columns))
192     optimized[optimized < 0] = 0
        optimized_out = (self.y_rescale*optimized).reshape((rows, columns))
194     return X_out, y_out, optimized_out
```

7.3 Aufbau und Funktion eines mehrschichtigen neuronalen Netzes

In Abschnitt 4.2.3 wurden schon die Grundprinzipien mehrschichtiger neuronaler Netze eingeführt. Die mathematische Realisierung wird im Folgenden allgemein anhand eines L-schichtigen neuronalen Netzes vorgestellt. Die Berechnung der Ausgabedaten (Klassifizierung) durch ein mehrschichtiges Netz wird durch die Vorwärtspropagation beschrieben. Die Eingabewerte propagieren von Schicht zu Schicht bis zur Ausgabeschicht. Äquivalent wird die Anpassung der Gewichte als Rückwärtspropagierung bezeichnet, da hierbei die Bewertung in der Ausgabeschicht zur Anpassung der Gewichte durch das Netz zurückgegeben wird. Beide Prozesse sind iterativ, es muss immer das gesamte Netz vor bzw. hinter einer Schicht durchlaufen werden, um die Aktivierung bzw. die vorzunehmende Anpassung der Gewichte in dieser Schicht zu berechnen.
Bei der Entwicklung der Algorithmen wurde das Buch *Machine learning mit Python und Scikit-learn und TensorFlow* von S. Raschka als Quelle und Orientierung verwendet [19].

Formale Einführung

Die Nomenklatur wird hier äquivalent zu den verschiedenen Ansätzen in Abschnitt 4.2.2 aufgebaut. Der Übersicht halber im Folgenden eine kurze Einführung.
Es wird ein Netz mit L Schichten betrachtet. Jede Schicht $l \in 0 \dots L$ hat d_l Knoten. Dies entspricht somit auch der Dimensionalität der Netto-Eingabe bzw. der Aktivierung der jeweiligen Schicht. Werden n Datensätze gleichzeitig verarbeitet, so können diese als Matrix $\mathbf{X} \in \mathbb{R}^{n \times d_0}$ eingegeben werden. Hierzu passen sich die Dimensionalitäten der Aktivierung und der Nettoeingabe in jeder Schicht zu $\mathbb{R}^{n \times d_l}$ an. Die Gewichte sind Matrizen $\mathbf{W}$ der Dimensionalität $d_l - 1 \times d_l$. Die Stapelverarbeitung (Änderung von n) hat keinen Einfluss auf die Dimensionalität der Gewichte.
Zur übersichtlichen Implementierung und breiten Anwendungsmöglichkeit des Netzwerks wird eine Klasse `NetWork` erstellt (vgl. die gesamte Klasse im A.2.8). Diese Klasse umfasst alle Funktionen, die zum Trainieren und Anwenden des neuronalen Netzes notwendig sind. In der `__init__` -Funktion werden die Parameter der Klasse initialisiert. Zwingend übergeben werden lediglich die Anzahl und die Größe der Schichten (`hidden_units`). `L2` stellt den Faktor in einer L2-Regularisierung nach Abschnitt 4.2.4 dar. `eta` ist die Lernrate, mit welcher die Gewichte aktualisiert werden. `cycles` stellt die Anzahl der im Training mit dem ganzen Datensatz durchzuführenden Zyklen dar, während `bathsize` die Anzahl der, je Aktualisierungszyklus der Gewichte, zu verarbeitenden Elemente angibt. `shuffle` definiert hierbei, ob die Elemente einer Teilmenge der `batchsize` in zufälliger Reihenfolge durch das Netz gegeben werden. Die Aktivierungsfunktion wird durch `act_func` ausgewählt. `seed` kann einen bestimmten Seed für die Erstellung der Zufallszahlen vorgeben.
Die einzelnen Parameter werden in den folgenden Funktionen zur Vorwärts- und Rückwärtspropagation sowie der Fit-Funktion benötigt.

```
class NetWork(object):
  """

  Parameters
  ----------
  hidden_units : array, shape = [n_layers]
    array of the number of units in each layer, at index 0 it is equal to n_features
  L2 : float (default: 0.)
    the factor for L2-weight-regularization
  eta : float (default: 0.001)
    the learning rate
  cylces : int (default 10)
    the number of cycles to train the weights
  batchsize : int (default 1)
    the number of samples to consider before adapting the weights
  act_func : str (default:'logistic')
    selects the activation function
  seed : give a special random seed to make outputs reproducable
  """
  def __init__(self, hidden_units = [], L2 = 0., eta = 0.01, cycles = 1000,
                                        batchsize = 1, shuffle = False, act_func =
                                         'logistic', seed = None):
    self.hidden_units = hidden_units
    self.L2 = L2
```

```
    self.eta = eta
    self.cycles = cycles
    self.batchsize = batchsize
    self.shuffle = shuffle
    self.act_func = act_func
    self.random = np.random.RandomState(seed)
```

7.3.1 Initialisierung der Gewichte

Die Gewichte werden als Matrizen mit der Dimension der vorangegangen Schicht und der Dimension der folgenden Schicht initialisiert. Als Startwerte werden kleine Zufallszahlen vergeben. Die Matrizen werden in einer Liste `W` für jede verdeckte Schicht und die Ausgabeschicht abgelegt. Das Bias `b` wird mit dem Wert 0 initialisiert. In die Implementierung ist das Konzept von S. Raschka eingeflossen [19, S. 400–405].

```
  def _initialize_weights(self):
    """give the weights-matrices the layer-dependant shape and initialze with small
                                        random(gauss) values

    Parameters
    ----------

    Returns
    -------
    self.W : list, shape = [n_layers]
      Gewichte, auf kleine Zufallszahlen initialisiert mit
      self.W[i] : array, shape[hidden_units[i], hidden_units[i+1]]
    self.b : list, shape = [n_layers]
      Offset, auf null initialisiert, mit
      self.b[i] : array, shape[hidden_units[i]]
    """
    self.W = []
    self.b = []
    for i in range(0, self.hidden_units.shape[0]-1):
      W_i = np.random.normal(loc=0.0, scale=0.1, size=(self.hidden_units[i], self.
                                        hidden_units[i+1]))   #weight matrices
                                        initialized with small random numbers
      b_i = np.zeros(self.hidden_units[i+1])    #bias always initialized with zeros
      self.W.append(W_i)
      self.b.append(b_i)
    return self
```

7.3.2 Vorwärtspropagation in der l-ten verdeckten Schicht

Die Nettoeingabe der verdeckten Schicht l mit d_l Knoten ergibt sich für eine n Elemente umfassende Schicht $l-1$ mit d_{l-1} Merkmalen aus ihrem Satz Gewichte $\mathbf{W}^{(l)} \in \mathbb{R}^{d_{l-1} \times d_l}$ und der Aktivierung der vorangegangenen Schicht $\mathbf{A}^{(l-1)} \in \mathbb{R}^{n \times d_{l-1}}$ zu

$$\mathbf{A}^{(l-1)}\mathbf{W}^{(l)} = \mathbf{Z}^{(l)} \in \mathbb{R}^{n \times d_l} \quad . \tag{7.6}$$

Äquivalent lässt sich mit mit $\mathbf{a}^{(l-1)} \in \mathbb{R}^{1\times d_{l-1}}$ auch elementweise die Nettoeingabe $\mathbf{a}^{(l-1)}\mathbf{W}^{(l)} = \mathbf{z}^{(l)} \in \mathbb{R}^{1\times d_l}$ bestimmen. Die Aktivierung ergibt sich zu

$$\mathbf{A}^{(l)} = \phi\left(\mathbf{Z}^{(l)}\right) \in \mathbb{R}^{n\times d_l} \quad . \tag{7.7}$$

Der Fall $l = 1$ entspricht der Nettoeingabe der Eingabeschicht und wird durch die Gewichte $\mathbf{W}^{(0)} = \mathbb{1}$ und $\mathbf{A}^{(0)} = \mathbf{X}$ definiert, wobei sich $\mathbf{X} \in \mathbb{R}^{n\times d_0}$ aus den n Merkmalsvektoren der Eingabedaten zusammensetzt.
Die Bias-Einheiten der verdeckten Schichten sind hier nicht explizit erwähnt. Diese lassen sich trivial direkt additiv hinzuzufügen oder durch Ergänzung einer Einheitsmatrix als $\mathbb{1} = \mathbf{a}_0^{(l-1)} \in \mathbb{R}^{n\times 1}$ und der Biaseinheit als $\mathbf{w}_0^{(l)} \in \mathbb{R}^{1\times d_l}$ realisieren [31, 19]. Im - im folgenden Abschnitt vorgestellten - Code wird das Bias additiv hinzugefügt.

Realisierung der Vorwärtspropagierung der Eingabewerte (Merkmale)

Die Realisierung der Vorwärtspropagierung geschieht in der Funktion `_forward_propagation()`. Als Eingabe dient die Matrix `X` der Merkmalsvektoren für jedes Element. Die Berechnung der Nettoeingabe jeder Schicht erfolgt als Matrixmultiplikation durch die Funktion `numpy.dot()` der vorangegangenen Aktivierung `A[-1]` mit dem Gewicht der i+1-ten Schicht `self.W[i]`. Die Bias-Einheit `self.b` wird nachträglich addiert. Die Aktivierung der aktuellen Schicht wird durch Anwendung der Aktivierungsfunktion in `_activation_function` bestimmt und an die Liste `A` angehängt.
Die Implementierung der Vorwärtspropagation ist dem Konzept von S. Raschka nachvollzogen [19, S. 400–405].

```
86  def _forward_propagation(self, X):
      """calculate the NetWorks's ouput by forward propagating the input values
                                          through each layer
88
      Parameters
90    ----------
      X : array, shape = [n_samples, n_features+1]
92      Array of input-values

94    Returns
      -------
96    A : list, shape = [n_layers]
        resulting values of the forward propagation with
98      A[i] : array, shape = [n_samples, hidden_units[i]]
      """
100   A = []    #initialize A as a list
      A.append(X)   #copy input values X as the first layer values
102   for i in range(0, self.hidden_units.shape[0]-1):
        Z = np.dot(A[-1], self.W[i]) + self.b[i]    #calculate the net-input for layer
                                          i+1 including the bias
104     A.append(self._activation_function(Z))    #calculate the activation
      return A
```

Die Funktion `_activation_function()` dient zur Auswahl und Berechnung der Aktivierung. Ein bei der Initalisierung der Klasse definierter Parameter `self.act_func` bestimmt die ange-

wendete Aktivierungsfunktion. Es sind zusätzlich zur logistischen Sigmoid-Funktion noch zwei weitere Funktionen definiert. Auf diese Funktionen wird nicht weiter eingegangen. Diese Funktionen kommen in dieser Arbeit nicht zur Verwendung.

```
34  def _activation_function(self, Z):
      """return the selected activation function
36
      Parameters
38    ----------
      Z : array, shape = [n_units]
40      Net input of the previous layer

42    Returns
      -------
44    phi : array, shape = [n_units]
        activation function of the net input
46    """
      if self.act_func == 'logistic':
48      """Compute logistic function"""
        phi = 1. / (1. + np.exp(-Z))
50      return phi
      elif self.act_func == 'softmax':
52      """Compute softmax function"""
        phi = np.exp(Z)/np.sum(np.exp(Z))
54      return phi
      elif self.act_func == 'tanh':
56      """Compute tangens hyperbolicus"""
        phi = np.tanh(Z)
58      return phi
      else:
60      print('ERROR: invalid activation function')
```

7.3.3 Rückwärtspropagierung der Abweichungen vom Ziel-Bild zur Optimierung der Gewichte

Um das Netz durch Anpassung der Gewichte zu trainieren, muss der Einfluss der einzelnen Gewichte jeder Schicht auf die endgültige Klassifizierung bestimmt werden. Hierzu wird ein Verfahren, welches bereits 1986 durch D. E. Rumelhart, G. E. Hinton und R. J. Williams veröffentlicht wurde, vorgestellt [31].
In Parallelität zu den Gleichungen (7.6) und (7.7) lassen sich die Berechnungen der Nettoeingabe $z_j^{(l)} \in \mathbb{R}$ sowie der Aktivierung $a_j^{(l)} \in \mathbb{R}$ für ein Element j der l-ten Schicht aufstellen

$$\mathbf{a}^{(l-1)}\mathbf{w}_j^{(l)} = \sum_{k=0}^{d_{l-1}} a_k^{(l-1)} w_{kj}^{(l)} = z_j^{(l)} \quad , \tag{7.8}$$

$$a_j^{(l)} = \phi\left(z_j^{(l)}\right) \quad . \tag{7.9}$$

Hierbei sind $\mathbf{a}^{(l-1)} \in \mathbb{R}^{1\times d_{l-1}}$ und $\mathbf{w}_j^{(l)} \in \mathbb{R}^{d_{l-1}\times 1}$, wenn stets alle Merkmale d_{l-1} der vorangegangen Schicht $l-1$ zur Berechnung eines Elementes der Schicht l herangezogen werden. d_l ist die Anzahl der Knoten in der l-ten Schicht.

Zur Bewertung einer Klassifizierung werden die Elemente $a_{j,i}^{(L)}$ der letzten Schicht L mit der jeweils angestrebten Klassifizierung $y_{j,i}$ wie in Abschnitt 4.2.2, Gleichung (4.4), verglichen. Es ergibt sich die Straffunktion

$$J_i\left(w_{pq}^{(l)}\right) = \frac{1}{2}\sum_{j=1}^{d_L}\left(y_{j,i} - a_{j,i}^{(L)}\left(w_{pq}^{(l)}\right)\right)^2 \quad \text{, mit} \quad l \in 1\dots L, p \in 1\dots d_{l-1}, q \in 1\dots d_l \quad . \tag{7.10}$$

Der Index i stellt das i-te Datenpaar des Trainingsdatensatzes dar, die Summation über j läuft über alle möglichen Klassifizierungen d_L der Ausgabeschicht. Es ist bereits die Abhängigkeit von allen Gewichten $w_{pq}^{(l)}$ eingefügt. p, q und l sind an dieser Stelle bewusst anders indiziert, da eine Abhängigkeit von jedem Element pq in jeder Schicht l vorhanden ist. Es existieren $d_{l-1} \times d_l$ Gewichte in jeder Schicht l, welche zusammengenommen die Matrix $\mathbf{W}^{(l)} \in \mathbb{R}^{d_{l-1}\times d_l}$ ergeben. Möchte man die Straffunktion mit dem Gradientenabstiegsverfahren bezüglich der Gewichte minimieren, werden die partiellen Ableitungen

$$\frac{\partial J}{\partial w_{pq}^{(l)}} \tag{7.11}$$

benötigt.
Das im Folgenden beschriebene Verfahren dient zur iterativen Bestimmung dieser partiellen Ableitungen und wird Backpropagation (Rückwärtspropagierung) genannt, da die Abweichung in der letzten Schicht (J_i) das Netz rückwärts durchläuft. Es wird die Indizierung über die Datenpaare (i) weggelassen, da zur Vereinfachung lediglich das Ergebnis eines Datenpaares betrachtet wird. Zunächst wird der erste Schritt ($L \to L-1$) betrachtet. Die Folgeschritte ($l \to l-1$) sind nur bei bereits erfolgtem vorangegangenem Schritt durchführbar. Wir schreiben zunächst

$$J\left(w_{pq}^{(L)}\right) = \frac{1}{2}\sum_{j=1}^{d_L}\left(y_j - a_j^{(L)}\left(z_j^{(L)}\left(w_{kj}^{(L)}\right)\right)\right)^2 \quad \text{, mit} \quad k, p \in 1\dots d_{L-1}, q \in 1\dots d_L \tag{7.12}$$

oder vereinfacht ohne sämtliche Abhängigkeiten

$$J\left(a_j^{(L)}\right) = \frac{1}{2}\sum_{j=1}^{d_L}\left(y_j - a_j^{(L)}\right)^2 \quad . \tag{7.13}$$

Des Weiteren wird als Aktivierungsfunktion die logistische Sigmoidfunktion (4.9) verwendet, sodass gilt

$$a_j^{(l)}(z_j^{(l)}) = \phi(z_j^{(l)}) = \frac{1}{1+e^{-z_j^{(l)}}} \quad . \tag{7.14}$$

Die Straffunktion lässt sich somit folgendermaßen differenzieren

$$\frac{\partial J}{\partial a_j^{(L)}} = a_j^{(L)} - y_j \quad . \tag{7.15}$$

Durch Ableitung der Sigmoidfunktion

$$\frac{\partial \phi(z)}{\partial z} = \phi(z)\left(1-\phi(z)\right) \quad \Leftrightarrow \quad \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} = a_j^{(l)}\left(1-a_j^{(l)}\right) \tag{7.16}$$

und Anwendung der Kettenregel ergibt sich

$$\frac{\partial J}{\partial z_j^{(l)}} = \frac{\partial J}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \quad . \tag{7.17}$$

Dies ist durch die Gleichungen (7.15) und (7.16) bereits für den Fall $l = L$ definiert. Erneute Anwendung der Kettenregel ergibt bei Ableitung der Berechnung der Nettoeingabe in Gleichung (7.8)

$$\frac{\partial J}{\partial w_{kj}^{(l)}} = \frac{\partial J}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{kj}^{(l)}} = \frac{\partial J}{\partial z_j^{(l)}} \, a_k^{(l-1)} \quad . \tag{7.18}$$

Auch dieser Zusammenhang ist bereits definiert, wenn Gleichung (7.17) gelöst und das Netz vorwärts durchlaufen und somit $a_k^{(l-1)}$ bekannt ist.
Für den Fall $l = L$ kann bereits das Differential (7.11) bestimmt werden. Um allerdings für alle $l \in 1 \dots L$ eine Lösung zu erhalten gilt es noch die Gleichung (7.15) für alle l zu bestimmen. Hier entsteht erneut durch Anwendung der Kettenregel der iterative Ansatz

$$\frac{\partial J}{\partial a_k^{(l-1)}} = \sum_{j=1}^{d_l} \frac{\partial J}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial a_k^{(l-1)}} = \sum_{j=1}^{d_l} \frac{\partial J}{\partial z_j^{(l)}} \, w_{kj}^{(l)} \quad , \tag{7.19}$$

welcher durch (7.8) und (7.17) definiert wird. (7.19) ersetzt somit (7.15) in (7.17) für alle Schichten $l < L$.
Zusammengefasst ergibt sich

$$\frac{\partial J}{\partial w_{kj}^{(l)}} = \frac{\partial J}{\partial z_j^{(l)}} \, a_k^{(l-1)} = \delta_j^{(l)} a_k^{(l-1)} \tag{7.20}$$

mit

$$\delta_j^{(l)} = \frac{\partial J}{\partial z_j^{(l)}} = \frac{\partial J}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} = \begin{cases} \left(a_j^{(l)} - y_j\right) \, a_j^{(l)} \left(1 - a_j^{(l)}\right) & , l = L \\ \left(\sum_{q=1}^{d_{l+1}} \delta_q^{(l+1)} \, w_{jq}^{(l+1)}\right) \, a_j^{(l)} \left(1 - a_j^{(l)}\right) & , l \in 0 \dots L-1 \end{cases} \tag{7.21}$$

als finale Rechenvorschrift. Nach dem Gradientenabstiegsverfahren werden die Gewichte mit der Lernrate $\eta > 0$ wie folgt aktualisiert [31, 19]

$$w_{pq}^{(l)} := w_{pq}^{(l)} - \eta \frac{\partial J}{\partial w_{pq}^{(l)}} \quad , \text{mit} \quad l \in 1 \dots L, p \in 1 \dots d_{l-1}, q \in 1 \dots d_l \quad . \tag{7.22}$$

Realisierung der Rückwärtspropagierung

Die Rückwärtspropagierung ist in der Funktion `_backward_propagation()` implementiert. Die Berechnung findet parallel zu den Gleichungen (7.20) und (7.21) statt und wird aufgeteilt zwischen der Ausgabeschicht und sämtlichen vorangegangenen Schichten. Es wird zunächst das

`delta` der Ausgabeschicht berechnet. Hierfür wird Gleichung (7.21) im Fall $l = L$ angewendet. Um den gesamten Datensatz verarbeiten zu können, werden Matrizen der Form $n \times d_l$ verarbeitet, wobei n die Anzahl der gleichzeitig verarbeiteten Datensätze und d_l die Anzahl der Knoten in Schicht l ist. Da das System rückwärts durchlaufen wird, müssen die Matrizen transponiert werden. Parallel zu Gleichung (7.20) wird die Aktivierung der vorangegangen Schicht `A[-2].T` auf `delta` angewendet. Über die Lernrate `self.eta` werden die Gewichte der Ausgabeschicht `self.W[-1]` aktualisiert.
Dieselben Schirtte finden für alle anderen Schichten mit der iterativen Berechnung von `delta` statt. Die Gewichte und die Lernrate sind Elemente der Klasse, da sie bei der nächsten Berechnung erneut benötigt werden.

```
  def _backward_propagation(self, A, y):
170   """backpropagate the error of the forward propagation to adapt the weights

172   Parameters
      ----------
174   A : list, shape = [n_layers]
        resulting values of the forward propagation with
176     A[i] : array, shape = [n_samples, hidden_units[i]]
      y : array, shape = [n_samples]
178     desired values

180   Returns
      -------
182   self.W : list, shape = [n_layers]
        updated set of weights
184     self.W[i] : array, shape[hidden_units[i-1], hidden_units[i]]
      """
186   delta = (A[-1].T-y)*A[-1].T*(1-A[-1].T)   #calculate delta for last layer
      dJdW = np.dot(A[-2].T, delta.T)   #calculate dJdW for last layer
188   self.W[-1] -= self.eta * dJdW   #adapt weights of the last layer
      for i in range(1,len(A)-1):
190     delta = np.sum(np.dot(self.W[-i], delta), axis = 0)*A[-(i+1)].T*(1-A[-(i+1)].T
                                          )   #iterative calculation of delta for
                                          layer i
        dJdW = np.dot(A[-(i+2)].T, delta.T)   #calculate dJdW for layer i
192     self.W[-(i+1)] -= self.eta * dJdW   #adapt weights of layer i
      return self
```

7.3.4 Fit-Methode

Die Gewichte des neuronalen Netzes werden in der Fit-Methode zunächst initialisiert und daraufhin angepasst. Es wird eine Ausgabematrix `A_out` mit den Dimensionen der Zielprojektion initialisiert. Über die Anzahl der durchzuführenden Iterationen (`cycles`) und die Trainingsbilder wird iteriert. Dabei wird jedes Trainingsbild in gegebenenfalls zufällige (`self.shuffle =True`) Teile der Größe `batchsize` geteilt. Jeder gleichzeitig zu verarbeitende Pixelstapel wird durch das Netz propagiert (`_forward_propagation()`). Das Ergebnis `A` wird mit mit dem Idealbild `y[picture_index, batch]` durch `_backward_propagation()` zur Adaption der Gewichte durch das Netz zurückgegeben. Nachdem alle Bilder durchlaufen sind, wird das Ausgabebild

A_out durch die predict()-Methode bestimmt und nach Durchlauf aller Zyklen zurückgegeben. predict() führt die Vorwärtspropagierung aus (siehe Anhang A.2.8)
Bei der Entwicklung der Fit-Methode wurden die Entwicklungen von S. Raschka als Orientierung verwendet [19, S. 400–405].

```
196   def fit(self, X, y):
        """pass data through network and adapt weights
198
        Parameters
200     ----------
        X : array, shape = [n_pictures, n_pixels, n_features+1]
202       training data
        y : array, shape = [n_pictures, n_pixels]
204     """
        self._initialize_weights()    #first, initialize the weights
206
        A_out = np.empty(y.shape)   #initialize the ouput matrix
208     for i in range(self.cycles):    #loop over the training cycles
          print('cycle:', i)
210       for picture_index in range(0, X.shape[0]):    #loop over all pictures
            pixel_index = np.arange(X.shape[1])   #get the indexes of the pixels to
212         if self.shuffle:
              np.random.shuffle(pixel_index)    #shuffle them
214         current_index = 0
            while current_index < (X.shape[1]-self.batchsize):    #go through it in
                                                  batchsized steps
216           batch = pixel_index[current_index:current_index + self.batchsize]
              current_index += self.batchsize
218           A = self._forward_propagation(X[picture_index, batch, :])   #forward the
                                                  batch
              self._backward_propagation(A, y[picture_index, batch])    #adapt the
                                                  weights
220         A_out[picture_index, :, None] = self.predict(X[picture_index,:,:])    #save
                                                  the predicted image
        return A_out
```

Es fällt auf, dass die Ausgabematrix A_out in jeder Iteration berechnet wird. Aus der Funktion zurückgegeben, wird sie allerdings erst nach Durchlauf aller Zyklen. Insofern wird die Matrix A_out in jeder Iteration überschrieben. In der aktuellen Implementierung dient dies keinem Zweck. Möchte man allerdings die Klassifizierung nach jeder Iteration prüfen, kann hier A_out in einer Variable gespeichert oder um eine Dimension der Gröse self.cycles erweitert werden, sodass alle Ausgabematrizen abgelegt werden können. Dann ist eine erweiterte Analyse des Fortschritts möglich.

7.4 Programm zur Merkmalerstellung und zum Trainieren des neuronalen Netzes

Um die Merkmale zu erstellen, die Datenmengen in Trainings- und Testdaten aufzuteilen, das Netz zu trainieren und die Hyperparameter zu variieren, dient das im Folgenden beschriebene PYTHON-Skript **seventh.py** (siehe auch A.2.6).

Die enthaltenen Funktionen `min_max_scale()` und `load_trackLength_filtered()` sind in Abschnitt 7.2.1 beschrieben. Die Klassen `data_matrices` (beschrieben in Abschnitt 6.2.2), `NetWork` (siehe Abschnitt 7.3) und `convolveWork` (dokumentiert in Abschnitt 7.2.2) werden eingebunden.
Die folgenden Zeilen laden die Projektionen der angegeben Bildgröße und definieren die Aufteilung in Test- und Trainingsdaten. Die `if`-Abfragen dienen hier dazu für den mit `dataset` gewählten Datensatz die Parameter richtig zu definieren. `selected_data_range` dient später zur richtigen Skalierung. Es folgt die Definition der Faltungsmatrizen, um die Merkmalsmatrizen zu erstellen. In `n_features_est` wird die Anzahl der erstellten Merkmale übergeben. Hier wird 4 zu der Anzahl der Faltungsmatrizen addiert, da dies der Anzahl der Pooling-Merkmale entspricht, die später im Code erstellt werden.

```
columns = 150
rows = 200

dataset = 1 #1,2,3,4
if ((dataset == 2 )):
  datanums = 1
  numinputdirs = 120
  num_trainsets = 100
  num_testsets = 20
elif ((dataset == 4 )|( dataset == 3)):
  datanums = 4
  numinputdirs = 120
  num_trainsets = 100
  num_testsets = 20
elif (dataset == 1):
  datanums = 1
  numinputdirs = 30
  num_trainsets = 25
  num_testsets = 5
base_inputdir = '/run/media/sj/Seagate Expansion Drive/Data_sjung/cumulated_' + str(
                                        dataset+1) + '/thirdCT_' + str(datanums) +
                                         '00000000_' + str(numinputdirs) + '
                                        slices_cumulated'

selected_data_range = 'uint16'

trackLength, trackLengthwindow = load_trackLength_filtered(columns, rows,
                                        numinputdirs, base_inputdir)    #read the
                                        test and training data

####### make the kernels for the convolutional features #######
edge_1 = np.array([[1,0,-1],[0,0,0],[-1,0,1]])
edge_2 = np.array([[0,1,0],[1,-4,1],[0,1,0]])
edge_3 = np.array([[-1,-1,-1],[-1,8,-1],[-1,-1,-1,]])
sharp_1 = np.array([[0,-1,0],[-1,5,-1],[0,-1,0]])
kernels = [np.ones((3,3)), np.ones((5,5)), edge_1, edge_2, edge_3, sharp_1]
n_features_est = len(kernels)+4
```

Um die Daten einzulesen und vorzubereiten, werden zunächst temporäre Arrays für die Eingabgedaten und die Zieldaten, `X_first` und `y_first` erstellt. In diese Arrays werden alle Pro-

jektionen und Zielprojektionen eingelesen. Die Zieldaten sind nicht das rauschreduzierte Bild (`trackLengthwindow[i].IDsPerPixel_matrix`), sondern die Differenz der rauschreduzierten und der originalen Projektion (`X_first[i] - trackLengthwindow[i].IDsPerPixel_matrix`. Die beiden temporären Arrays werden an ein `convolveWork`-Objekt übergeben, um den Datensatz zu erstellen, welcher durch das Netz verarbeitet werden kann. Als Merkmale (`features`) werden zunächst die Pooling-Merkmale erstellt und dann, mit den zuvor definierten Faltungsmatrizen, die weiteren Merkmalsmatrizen berechnet. Dadurch ist die Anzahl der Merkmale `n_features` definiert und die für das Netz verarbeitbaren Matrizen können mit der `vectorize`-Funktion erstellt werden.

```
116 ####### LOAD & PREPARE TRAINDATA #######
    X_first = np.empty((num_trainsets, rows, columns))    #initialize original
                                                          projection array
118 y_first = np.empty((num_trainsets, rows, columns))    #initialize desired projection
                                                           array
    X_train = np.empty((num_trainsets, rows*columns, n_features_est+1))   #initialize
                                                          the training data arrays
120 y_train = np.empty((num_trainsets, rows*columns))

122 makefeatures_train = []
    for i in range(0,num_trainsets):
124   X_first[i] = trackLength[i].IDsPerPixel_matrix    #load the original projections
      y_first[i] = X_first[i] - trackLengthwindow[i].IDsPerPixel_matrix   #load the
                                                          filtered, desired projections
126   makefeatures_train.append(convolveWork(X_first[i], y_first[i]))   #initialize the
                                                          makefeatures object to make the features
      makefeatures_train[i].features = [makefeatures_train[i].maxpool(),
                                                          makefeatures_train[i].minpool(),
                                                          makefeatures_train[i].meanpool(),
                                                          makefeatures_train[i].stdpool()]    #
                                                          append the standart features
128   for j in range(len(kernels)):
        makefeatures_train[i].features.append(makefeatures_train[i].conv_w_kernel(
                                                          kernels[j]))    #append the convolutional
                                                          features
130   n_features = len(makefeatures_train[i].features)    #number of features
      if n_features != n_features_est:
132     print('ERROR:WRONG n_feature INITIALIZATION')
      X_train[i,:,:], y_train[i,:], X_train_rescale, y_train_rescale= makefeatures_train
                                                          [i].vectorize(set_scale = True)   #get the
                                                           vectorized training data for the NetWork
```

Die Datensätze für Test- und Validierungsdaten werden auf die gleiche Weise erstellt, hierbei kann direkt mit dem bekannten Parameter `n_features` initialisiert werden (siehe hierzu das vollständige Skript im Anhang A.2.6).

Für das neuronale Netz müssen einige Parameter initialisiert werden. Zunächst wird die Anzahl der Trainingszyklen `n_cycles` definiert. Um bessere Optimierungsmöglichkeiten für die weiteren Hyperparameter zu haben, wird für jeden Hyperparameter eine Liste mit möglichen Werten erstellt. `n_hidden` gibt die Anzahl der Knoten innerhalb der verdeckten Schicht vor. In der aktuellen Konstellation wird hierüber nur die Anzahl der verdeckten Elemente in einer Schicht

definiert. Das Netz kann auch mit mehreren Schichten betrieben und initialisiert werden. Dies ist allerdings nicht im Hauptprogramm implementiert. Aus `etas` lässt sich eine Lernrate mit einem Wert zwischen 0.001 und 0.1 auswählen. `batchsizes` bietet die Möglichkeit, zwischen pixelweiser und bildweiser Verarbeitung der Daten, eine sinnvolle Größe des Datensatzses zu ermitteln, um das Netz zu trainieren. Nach der Definition der Parameter als Listen, wird jeweils der Index eines Standard-Wertes definiert, da nur ein Parameter variiert werden kann. Die Standardwerte sind auf `eta`=0.001, `batchsize`=330 (später im Programm wird 1 addiert) und `n_hidden`=10 gesetzt. Die Parameter werden in einer Liste zusammengefasst, aus welcher dann ausgewählt werden kann, welche Parameter konstant bleiben und welcher variiert wird. Die Möglichkeit, die Hyperparameter einzeln zu variieren, dient dazu, hier die optimalen Werte zu finden. Ist eine Variation gewünscht, wird der `do_hyparmod`-Parameter auf `True` gesetzt und mit `modpar` der zu variierende Parameter ausgewählt. Es kann stets nur ein Hyperparameter variiert werden. Die `cost_` und `cost_L2_`-Arrays werden erstellt, um später die cost-Funktion bzgl. des variierten Parameters zu speichern (siehe hierzu die Straffunktion (cost) in den Abschintten 4.2.2 und 4.2.2 und die L2 Regularisierung in Abschnitt 4.2.4 sowie die Implementierung dieser Funktionen in der NetWork-Klasse in Abschnitt A.2.8). Die Bedeutung der cost-Funktionen ist in Abschnitt 4.2.4 erläutert. Anhand der cost-Funktionen kann später die optimale Konstellation erörtert werden.

```
######## CALCULATE & OPTIMIZE WEIGHTS ########
n_cycles = 5      #how many cycles the NetWork will be trained
n_output_features = 1   #one means, that one pixel with multiple features in the
                                        input layer will result in one pixel in
                                        the output layer
n_hidden = np.arange(1,100, 1)    #n_features + 1
etas = np.arange(0.001, 0.1, 0.01)
batchsizes = np.arange(0,rows*columns,30)[1:]
fixed = np.array([1])   #select if no hyperparameter modulation is wanted

parameters = [n_hidden, etas, batchsizes, fixed]
parameter_labels = ['n_hidden', 'etas', 'batchsizes', 'None']

eta_index = 0
batchsize_index = 10
hidden_unit_index = 10  # hier w"are anstatt einer zahl ein array 'beliebiger' form
                                        denkbar, um auch mehrere schichten zu
                                        ermoeglichen


do_hyparmod = False

indexes = [hidden_unit_index, eta_index, batchsize_index, 1]
if do_hyparmod:
  modpar = 1  #modulate parameter number  0:n_hidden, 1:etas, 2:batchsize, 3:no
                                        hyperparameter modulation
  cost_train = np.empty(parameters[modpar].shape[0])
  cost_L2_train = np.empty(parameters[modpar].shape[0])
  cost_test = np.empty(parameters[modpar].shape[0])
  cost_L2_test = np.empty(parameters[modpar].shape[0])
else:
  modpar = 3
```

Für den Fall, dass keine Modulation stattfindet, wird `modpar`=3 gesetzt, da in den Listen der Parameter je an vierter Position der Fall keiner Variation abgedeckt ist.
Wird ein Hyperparameter variiert, wird über alle Möglichkeiten dieses Parameters iteriert. Die einzelnen Parameter werden an das `NetWork`-Objekt `Net1` übergeben. Die `NetWork.fit()`-Funktion führt die Anpassung der Gewichte durch und gibt das Ergebnis an `optimized_train` aus. Das Ergebnis umfasst eine Projektion mit den bestimmten absoluten Rauschanteilen je Pixel und kann somit mit `y_train` verglichen werden.

```
178 ####### ITERATE OVER HYPERPARAMETERS #######
    for m in range(0, parameters[modpar].shape[0]):
180   print('Iteration:', m, '/', parameters[modpar].shape[0])
      indexes[modpar] = m
182   print('layers:',np.array([n_features+1, parameters[0][indexes[0]],
                                 n_output_features]))
      print('eta:', parameters[1][indexes[1]])
184   print('cycles:', n_cycles)
      print('batchsize:', parameters[2][indexes[2]])
186
      Net1 = NetWork(hidden_units = np.array([n_features+1, parameters[0][indexes[0]],
                                 n_output_features]), eta = parameters[1][
                                 indexes[1]], cycles = n_cycles, batchsize
                                 = parameters[2][indexes[2]], shuffle =
                                 False, seed = 1)   #initialize the NetWork
188   optimized_train = Net1.fit(X_train, y_train)    #fit the data and return the
                                 optimized pictures for the training data
```

Zur Bewertung der Resultate müssen sowohl das vorhergesagte Streubild (`optimized_train`), als auch das bekannte Streubild (`y_train`) und die Eingabedaten (`X_train`) zurückskaliert werden. Des Weiteren können die cost-Funktionen bestimmt werden. Die gleichen Schritte werden auch für die Testdaten durchgeführt. Hier muss zunächst noch das trainierte Netz auf die Daten angewendet werden. Dies geschieht durch die `predict()`-Funktion, welche eine Vorwärtspropagierung durchführt (vgl. das die NetWork-Klasse in Abschnitt A.2.8). Durch die `for`-Schleife über einen der Hyperparameter werden das Netz mehrfach angepasst und jeweils die cost-Werte der Trainings- und Testdaten gespeichert.

```
190   ####### CALCULATE RESULTS #######
      X_train_out = np.empty((X_train.shape[0], rows, columns))   #initialize the output
                                                                   arrays
192   y_train_out = np.empty((y_train.shape[0], rows, columns))
      optimized_train_out = np.empty((y_train.shape[0], rows, columns))
194   if do_hyparmod:
        cost_train_temp = 0
196     cost_L2_train_temp = 0
      for j in range(0,X_train.shape[0]):   #make the output arrays from the network-
                                              output
198     X_train_out[j,:,:], y_train_out[j,:,:], optimized_train_out[j,:,:] =
                                              makefeatures_train[j].rescale_and_reshape(
                                              X_train[j,:,0], y_train[j,:],
                                              optimized_train[j,:], rows, columns)
        if do_hyparmod:
200       cost_train_temp += Net1.cost_function(optimized_train, y_train[j,:])
          cost_L2_train_temp += Net1.cost_L2(optimized_train, y_train[j,:])
202   if do_hyparmod:
        cost_train[m] = cost_train_temp/X_train.shape[0]
204     cost_L2_train[m] = cost_L2_train_temp/X_train.shape[0]

206   ####### CALCULATE RESULTS #######
      X_test_out = np.empty((X_test.shape[0], rows, columns))
208   y_test_out = np.empty((y_test.shape[0], rows, columns))
      optimized_test_out = np.empty((y_test.shape[0], rows, columns))
210   if do_hyparmod:
        cost_test_temp = 0
212     cost_L2_test_temp = 0
      for j in range(0,X_test.shape[0]):
214     optimized_test = Net1.predict(X_test[j,:,:])
        X_test_out[j,:,:], y_test_out[j,:,:], optimized_test_out[j,:,:] =
                                              makefeatures_test[j].rescale_and_reshape(
                                              X_test[j,:,0], y_test[j,:], optimized_test
                                              , rows, columns)
216     if do_hyparmod:
          cost_test_temp += Net1.cost_function(optimized_test, y_test[j,:])
218       cost_L2_test_temp += Net1.cost_L2(optimized_test, y_test[j,:])
      if do_hyparmod:
220     cost_test[m] = cost_test_temp/X_test.shape[0]
        cost_L2_test[m] = cost_L2_test_temp/X_test.shape[0]
```

Nach dem mehrfachen Trainieren und Testen des Netzes wird das Netz noch auf einen Validierungsdatensatz angewendet. Dieser wird nach dem gleichen Verfahren wie die beiden anderen Datensätze eingelesen und verarbeitet. Daraufhin wird eine Log-Datei mit allen gesetzten Parametern geschrieben und die Projektionen, original, durch das Netz gefiltert und aus der Simulation vorhergesagt, werden ausgegeben. Die cost-Werte werden geplottet und gespeichert. Details hierzu sind im Skript in Abschnitt A.2.6 nachzulesen.

8 Anwenden des neuronalen Netzes

Die Anwendung des neuronalen Netzes erfolgt nach der umfangreichen Erstellung und Vorverarbeitung der Simulationsdaten. Um die Übersicht zu erhalten und ein Verständnis für die Notwendigkeit der bisher vorgestellten Algorithmen zu schaffen, wird zunächst der Ablauf von der Simulation zur Anwendung des neuronalen Netzes beschrieben.

8.0.1 Workflow von der Simulation zum neuronalen Netz

Äquivalent zu dem Makro-Skript, welches in Abschnitt 5 vorgestellt wird (gesamte Skripte im Anhang unter A.3), werden zunächst mehrere (funktional) identische Skripte erstellt. Diese können parallel auf einem Rechencluster GATE-Simulationen durchführen. Die Simulationen erzeugen jeweils eine reduzierte Anzahl Photonen, welche das Objekt unter allen vorgegebenen Winkeln abbilden (siehe Hierzu Abschnitt 5.4). Jeder parallel simulierte Datensatz umfasst infolgedessen für jede Objektorientierung (Winkel) eine Projektion geringer Intensität sowie einen ROOT-Tree. In ROOT sind die Winkel-Projektionen durch ihren Zeitstempel unterscheidbar.
Das Skript **`pythonrootv2.py`** (siehe Abschnitt 6.2.1 sowie das Skript im Anhang A.2.2) liest die Projektionsdaten sowie die für die Filterung relevanten Werte aus dem ROOT-Tree jeder Simulation aus und speichert diese als `numpy`-Arrays für jeden Tree und Winkel. Diese Arrays können dann durch das Skript **`adder.py`** (erläutert in Abschnitt 6.3 und einzusehen im Anhang A.2.4) für jeden Winkel aus allen parallelisierten Simulationen zusammengefasst werden. **`filter_by_trackLength.py`** (siehe A.2.5) erstellt aus den kumulierten Arrays die nach der Weglänge gefilterten Ground Truth-Daten für jede Objektorientierung. Die Filterung ist in Abschnitt 6.4.3 erläutert.
Daraufhin kann das Hauptprogramm (**`seventh.py`**) (siehe A.2.6), vorgestellt in Abschnitt 7, die Merkmale erstellen, das neuronale Netz initialisieren und an die Daten anpassen.

8.0.2 Anwendung

Das neuronale Netz wird auf die in Abschnitt 5.4 vorgestellten Phantome angewendet. Stehen 30 Projektionen zur Verfügung, werden 25 als Trainingsdaten und 5 als Testdaten verwendet. Stehen 120 Projektionen zur Verfügung findet eine Aufteilung 100 zu 20 statt. Es wird bei jedem Training der gleiche Validierungsdatensatz, bestehend aus 8 Projektionen, verwendet.
Die Hyperparameter werden bei ihren Standardwerten belassen (vgl. Abschnitt 7.4). Eine Variation der Hyperparameter zur Ermittlung optimaler Werte wurde angestrebt, konnte aber zeitlich nicht realisiert werden.

8.1 Trainieren des Netzes

Das Netz wird trainiert, indem in jedem Zyklus die Trainingsdaten durch das Netz gegeben und nach jedem Batch (siehe `batchsize` Abschnitt 7.3.4) die Gewichte aktualisiert werden. Die Anzahl der Zyklen hat hierbei erheblichen Einfluss auf die spätere Anwendbarkeit und Güte des Netzes. Es kann analysiert werden nach welcher Anzahl Zyklen die Anpassung des Netzes an die

Daten nicht weiter steigt. Hierfür wird zunächst die Anwendung auf die Trainingsdaten selbst und im nächsten Schritt die Anwendung auf Test- bzw. Validierungsdaten (vgl. Abschnitt 8.2 und 8.3) betrachtet. Es können periodisch unterschiedliche Grade der Anpassung auftreten. Das heißt, dass sich Zustände des Netzes (die Werte der Gewichte) abhängig von den zuletzt verarbeiteten Projektionen wiederholen.
Das Netz wird nicht auf periodische Anpassungen untersucht. Die Bewertung der Anpassung findet rein optisch statt. Betrachtet werden hier beispielsweise das vorhandene Rauschen und die Detailauflösung. Hauptsächlich wird der Vergleich zu den, der Simulation entstammenden angestrebten Projektionen und Streumatrizen herangezogen. Quantitative Bewertungen anhand eines Signal/Rausch-Verhältnisses, der Entropie, quantifizierbarer Detailauflösung oder einer Frequenzuntersuchung im Fourierraum sind denkbar und sinnvoll, werden aber im Rahmen dieser Arbeit nicht durchgeführt.

8.1.1 Eigenschaften zur Qualitätsbewertung

Die verschieden gefilterten Projektionen werden rein optisch durch Betrachtung bewertet. Hierbei wird auf bestimmte Details geachtet, die eine Einordnung und Bewertung erleichtern. Um die Erkennbarkeit von relevanten Details zu ermöglichen, wird bei Matrizen mit wenigen, niedrigen Grauwerten die invertierte Darstellung gewählt.
Sämtliche Phantome weisen Materialgrenzen auf. Diese Grenzen stellen sich in der Röngenprojektion als Übergang zwischen unterschiedlich starker Absorption und somit verschiedenen Grauwerten dar. Insofern sind derartige Übergänge auch in den Phantomen, welche lediglich aus Aluminium bestehen, aufgrund von unterschiedlichen Materialdicken beobachtbar. Diese Materialgrenzen sollten mögichst gut erkennbar sein und es sollte eine klare Unterscheidung zwischen verschiedenen Materialien (oder Materialdicken) anhand des Grauwertes möglich sein.
Die Qualität der Projektion und somit der Filterung wird auch anhand der Auflösung der Objekte mit feinen Strukturen bewertet. Die Kugeln weisen grobe Strukturen und somit geringe Raumfrequenzen auf, während die Φ's kleine Strukturen und somit hohe Raumfrequenzen haben. Die Strahlung, die durch die Öffnungen in den Buchstaben auf den Detektor trifft, ist aufgrund der geringen Größe des abzubildenden Details schwer von einem auf der Streustrahlung basierenden Artefakt zu unterscheiden. Eine Filterung kann also die Auflösung der Buchstaben verbessern oder auch verschlechtern.
Allgemein können durch Streustrahlung, welche mehrere benachbarte Pixel trifft, Artefakte entstehen. Diese Artefakte zu filtern ist eine besondere Herausforderung, die andere Anforderungen stellt als Streustrahlung in einem einzelnen Pixel in streustrahlungsfreier Umgebung herauszufiltern.
Neben den Projektionen und Ausschnitten der Projektionen kann auch direkt die durch das Netz ermittelte Streustrahlung betrachtet werden. Die Darstellung der Streustrahlung eignet sich gut um hohe Anteile fälschlicherweise als Streustrahlung definierter Pixelwerte zu erkennen. Ist in dem Streubild Objektinformation zu erkennen, so werden signifikante Teile der Projektion gefiltert, welche zur relevanten Bildinformation beitragen.
Da die Streumatrizen nur sehr geringe Werte im Vergleich zur gesamten Projektion beeinhalten, werden die Streumatrizen anders skaliert. Zunächst bleibt es wichtig, dass alle zu vergleichenden Matrizen gleich skaliert werden. Deswegen werden die Streumatrizen im Folgenden zunächst

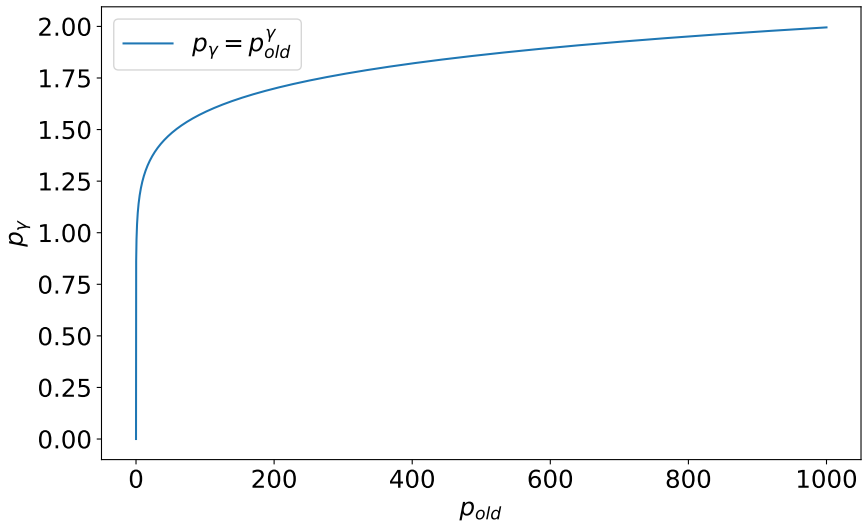


Abbildung 8.1 Plot der Funktion zur sogenannten Gammakorrektur nach Gleichung 8.1 für $\gamma = 1$.

relativ zum Maximalwert der simulierten Streumatrix (Ground Truth) skaliert. Aufgrund der geringen Dynamik, die die Streumatrizen umfassen sind bei linearer Skalierung trotz Anpassung des Wertebereichs durch den geringen Kontrast nur wenige Details erkennbar. Infolgedessen werden die Streumatrizen nicht-linear skaliert.
Es werden die Grauwerte durch eine Potenzfunktion auf ein neues Mapping übertragen. Dieses Konzept ist allgemein bekannt als Gamma-Korrektur, da die Potenz γ hier der einzige Parameter ist, welcher vorzugeben ist. Gleichung (8.1) zeigt formal dieses Mapping für Grauwerte p. In der Darstellung der Streumatrizen wird im Folgenden $\gamma = 0.1$ verwendet. Die entsprechende Funktion ist in Abbildung 8.1 dargestellt [30, S. 40].

$$p \to p^\gamma \tag{8.1}$$

Jedes Bild beeinhaltet sogenanntes Rauschen. Gemeint ist hiermit in der Regel das statistische Schwanken eines Pixelwertes um den jeweiligen Erwartungswert. Da diese Schwankung zunächst in jedem Pixel unabhängig von der Umgebung auftritt, unterscheiden sich die Grauwerte eines Bereiches gleicher Absorption. Auch die Streustrahlung erzeugt Bildrauschen. Aufgrund des (in den durchgeführten Simulationen) geringen Anteils der Streustrahlung gegenüber der ungestreuten Strahlung und der nahezu isotropen Verteilung der Streustrahlung, weist die Streustrahlung eine deutlich geringere Raumfrequenz auf als das statistische Bildrauschen. Das heißt, dass das statistische Bildrauschen in der Größenordnung der Pixel zu beobachten ist, während das Bildrauschen, welches auf Streustrahlung zurückzuführen ist, eine deutlich geringere Frequenz aufweist und somit erst bei Betrachtung eines Bereiches vieler Pixel zu charakterisieren ist. Um das statistische Rauschen absolut zu charakterisieren, kann eine Überschlagsrechnung angestellt werden. Jede Projektion wird mit 3333333 Photonen bestrahlt (vgl. Abschnitt 5.4, Ausnahme ist Datensatz II). Da jede Projektion aus 150×200 Pixeln besteht, lässt sich die Anzahl Teilchen pro Pixel auf 111 genähert mitteln (alle Teilchen treffen den Detektor). Hier wird davon ausgegangen, dass sich kein absorbierendes Objekt im Strahlengang befindet und die Photenen isotrop über den Raumwinkelbereich des Detektors verteilt sind. Der Fehler eines Mittelwerts n kann mit

$\sqrt{n}$ abgeschätzt werden. Bei dieser groben Überschlagsrechnung erhält man ein rein statistisches Rauschen von $\sqrt{111} \approx 11$. Dies entspricht der Größenordnung des verbliebenen Rauschens in den nach der Simulation gefilterten Projektionen. Zur genaueren Bewertung sollte natürlich der Wert $\sqrt{n}$ für die beobachtete Bildregion bestimmt werden.

Welcher Teil des Rauschens Streustrahlung ist, wird durch die Filterung auf Basis der Weglänge in den Simulationsdaten abgeschätzt. Diese Information wird hier als *Ground Truth* verwendet. Auf Basis dieser Phänomene wird im Folgenden die Leistung des erstellten neuronalen Netzes bei unterschiedlichen Datensätzen im Training und unterschiedlicher Anzahl Trainingsepochen untersucht.

8.1.2 Datensatz III - Trainingsdaten

Das Training mit Datensatz III wird dazu verwendet, zunächst eine geeignete Anzahl durchzuführender Iterationen zu definieren. Die übrigen Datensätze werden auf dieser Basis analysiert.

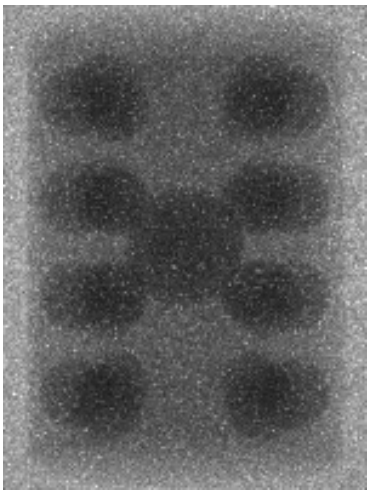
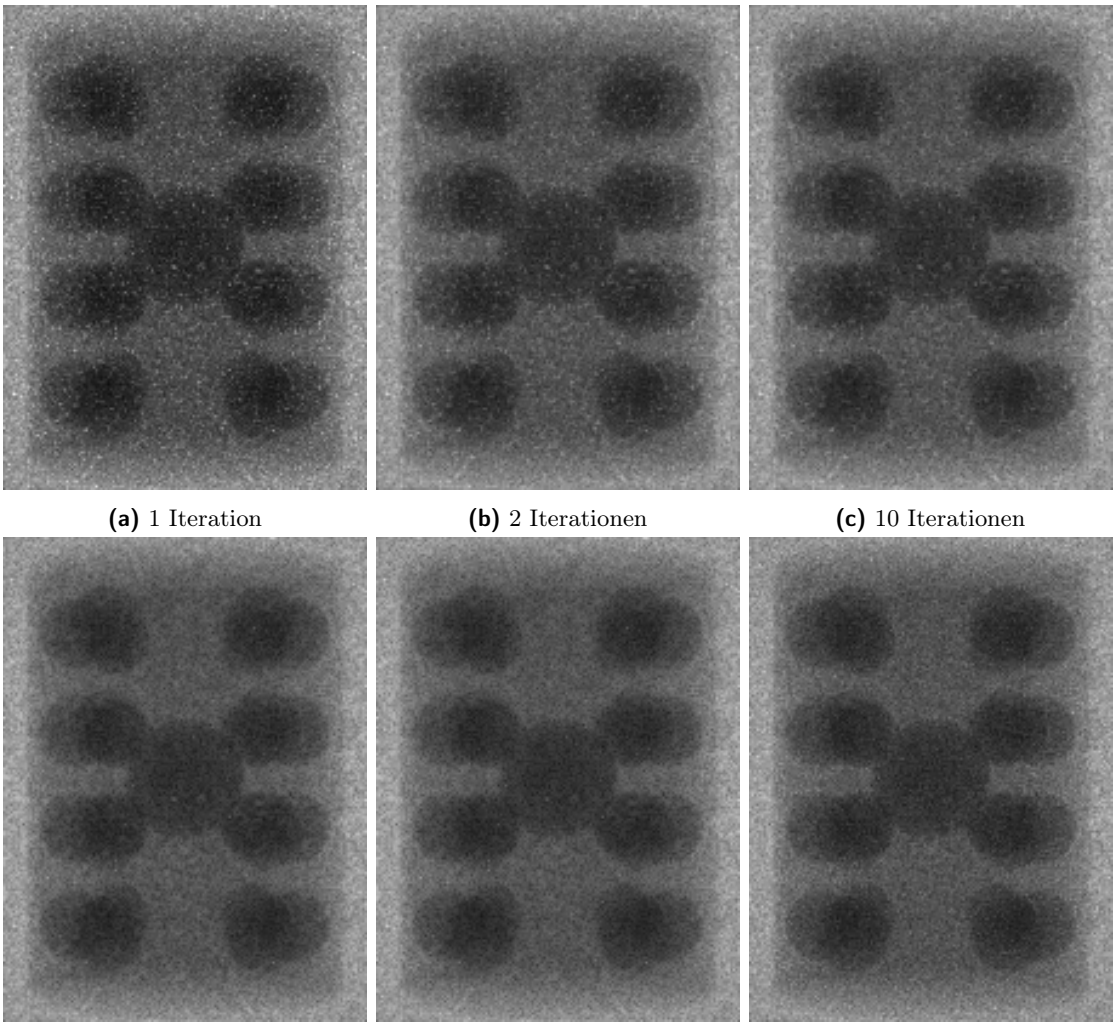


Abbildung 8.2 Eine ungefilterte Projektion des Datensatzes III, anhand derer im Folgenden das Training des Netzes beobachtet wird.

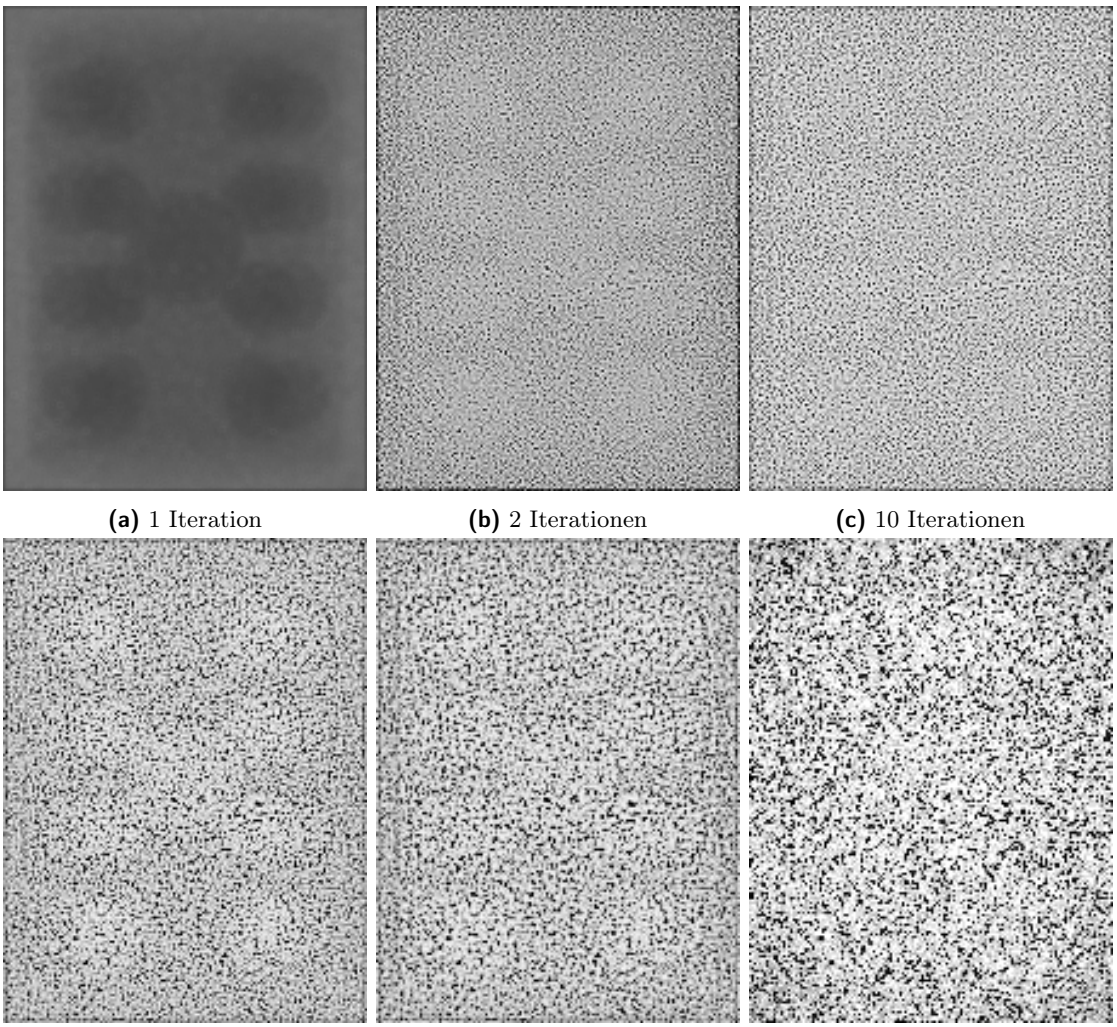
Zunächst wird das Netz über eine variable Anzahl Zyklen mit den Trainingsdaten trainiert. In Abbildung 8.3 ist der Verlauf des Trainings zu erkennen. 8.2 zeigt die Ausgangsprojektion aus Datensatz III, während 8.3f die angestrebte, rauschfreie Projektion darstellt. 8.3a bis 8.3e zeigen die Anpassung nach 1, 2, 10, 100 und 1000 Iterationen, in denen die Gewichte des Netzes optimiert werden. Es fällt auf, dass die größte Verbesserung direkt nach der zweiten Iteration stattfindet. Erst 100 Iterationen zeigen hier wieder eine deutliche Veränderung. 1000 Iterationen können weitere Artefakte herausfiltern (siehe mittlere Kugel, Ausschnitt in Abbildung 8.5). Es bleibt allerdings ein deutlicher Unterschied zur Ground Truth vorhanden.
In Abbildung 8.4 ist das, durch das Netz bestimmte, Rauschen nach den gleichen Iterationszahlen dargestellt. Das Rauschen ist anhand des erwarteten Rauschbildes 8.4f skaliert, invertiert sowie mit $\gamma = 0.1$ potenziert. Andernfalls wären nahezu keine Grauwerte erkennbar (vgl. A.1 und A.2 in Abschnitt A.1.1). Es lassen sich die obigen Aussagen bestätigen. Die größte Anpassung findet mit der 2. Iteration statt, während ein nächster großer Unterschied zwischen 10 und 100 Iterationen festzustellen ist. 1000 Iterationen erzeugen nur noch einen minimalen Unterschied durch



(a) 1 Iteration **(b)** 2 Iterationen **(c)** 10 Iterationen

(d) 100 Iterationen **(e)** 1000 Iterationen **(f)** Ground Truth

Abbildung 8.3 Bild (a), (b), (c), (d) und (e) zeigen die Projektionen (8.2) des Datensatzes III nach 1, 2, 10, 100 und 1000 Iterationen des neuronalen Netzes. (f) zeigt die als rauschfrei angenommene Projektion, die zum Training verwendet wird.



(a) 1 Iteration **(b)** 2 Iterationen **(c)** 10 Iterationen

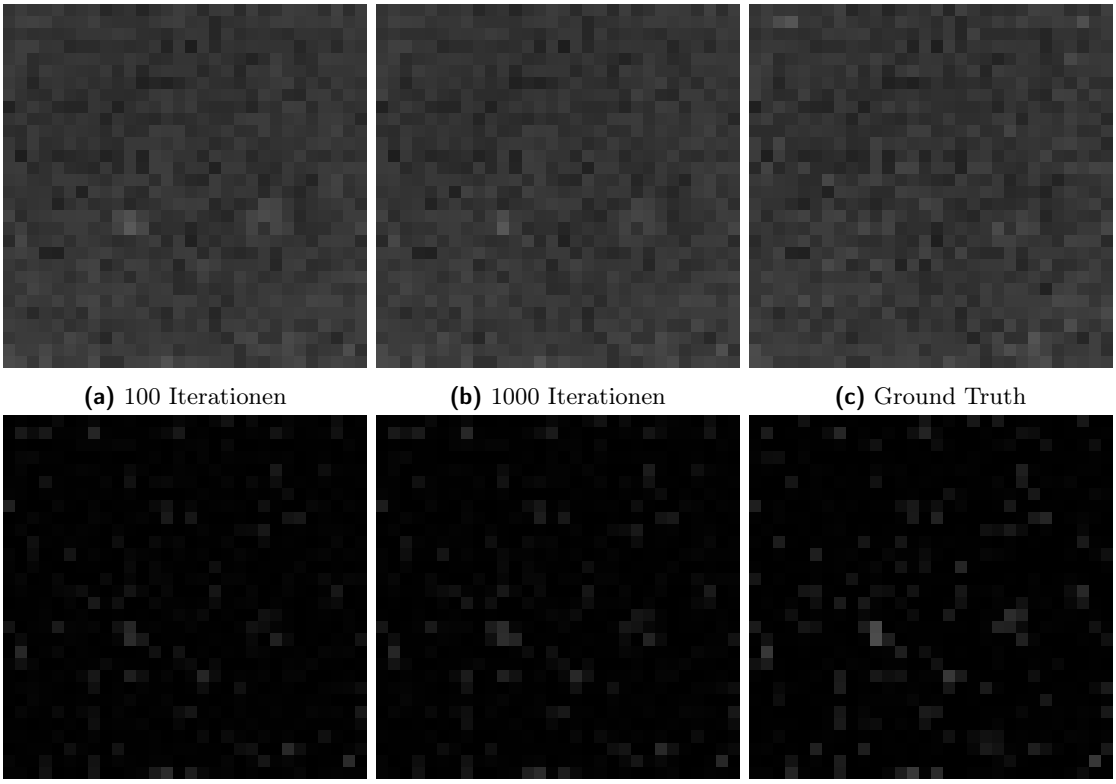
(d) 100 Iterationen **(e)** 1000 Iterationen **(f)** Ground Truth

Abbildung 8.4 Bild (a), (b), (c), (d) und (e) zeigen den bestimmten Rauschanteil der Projektion (Datensatz III) nach 1, 2, 10, 100 und 1000 Iterationen des neuronalen Netzes. (f) zeigt den, aus den Simulationsdaten vorhergesagten Rauschanteil. Die Abbildung zeigt Rauschanteile relativ zum angestrebten Bild (f) skaliert. Die unskalierte Abbildung ist im Anhang unter A.1 in Abschnitt A.1.1 einzusehen. Des Weiteren wird mit $\gamma = 0.1$ eine Gamma-Korrektur vorgenommen. Der Einfluss davon kann durch Vergleich mit Abbildung A.2 im Anhang betrachtet werden. Die Darstellung verwendet die invertierten Grauwerte, um einen größeren Kontrast zu bieten.

die reduktion des hochfrequenten Rauschens. Im Vergleich zu der angestrebten Streustrahlung sind die Pixelwerte nach 1000 Iterationen zu gering und haben eine zu große Raumfrequenz, um das Rauschen vollständig zu korrigieren.

Zur Bewertung wird auch die Darstellung der Differenz (betragsmäßig) zwischen gefilterter und Ground Truth-Projektion (bzw. gefiltertem und angestrebtem Rauschen) betrachtet. Diese Abweichung wird ebenfalls skaliert und mit $\gamma = 0.1$ potenziert, damit Pixelwerte erkennbar sind. Für die Skalierung wird der Einfachheit halber ebenfalls das Maximum der erwarteten Streustrahlung verwendet. In Abbildung 8.6 ist dies für die gefilterte Projektion nach 2 bzw. 1000 Trainingszyklen dargestellt.

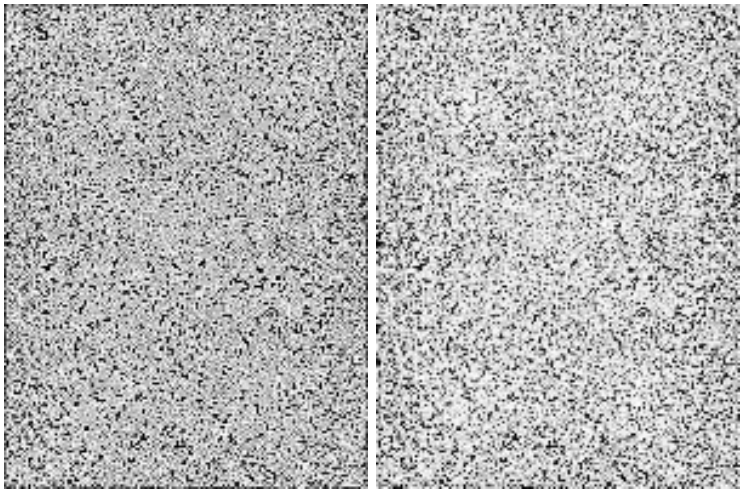
Nach 2 Iterationen sind die Abweichungen hochfrequenter und somit homogener verteilt als nach 1000 Iterationen. Es spiegeln sich hier also die Erkenntnisse der vorangegangenen Beobachtungen wieder. Es bleibt auch nach 1000 Projektionen eine erkennbare Abweichung von der



(a) 100 Iterationen **(b)** 1000 Iterationen **(c)** Ground Truth

(d) 100 Iterationen **(e)** 1000 Iterationen **(f)** Ground Truth

Abbildung 8.5 Die Bilder (a), (b) und (c) zeigen je den gleichen Ausschnitt der Projektionen 8.3d, 8.3e und 8.3f aus Datensatz III und (d), (e) und (f) zeigen die entsprechenden Rauschbilder. Links unterhalb der Mitte ist ein Artefakt, das durch 1000 Iterationen weiter reduziert werden kann als durch 100 Iterationen.



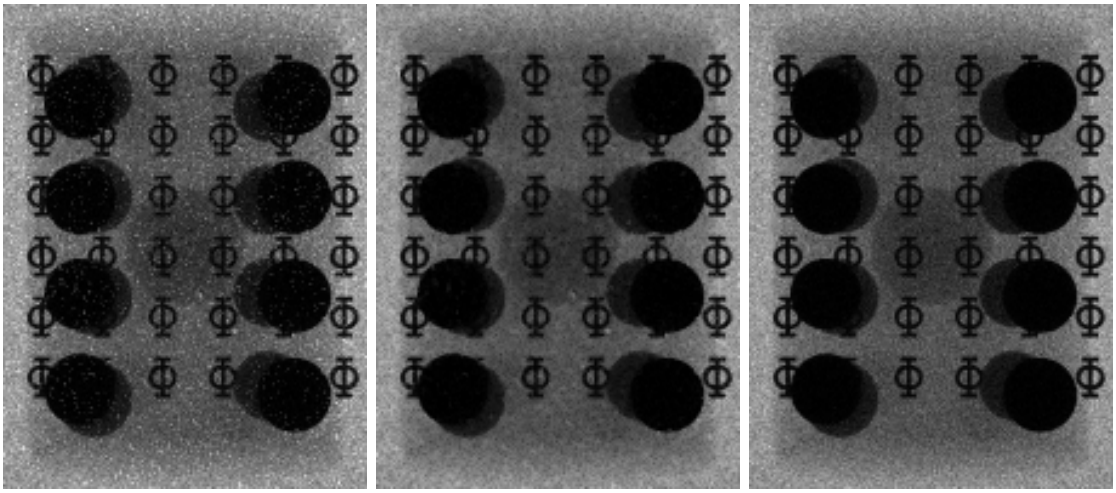
(a) 2 Iteration **(b)** 1000 Iterationen

Abbildung 8.6 In den Abbildungen (a) und (b) sind die absoluten Differenzen zwischen der Filterung und Ground Truth nach 2 und 1000 Iterationen zu sehen (Projektion aus Datensatz III). Beide Abbildungen sind gleichermaßen, relativ zum Maximum der angestrebten Streustrahlung (vgl. Abbildung 8.4f), hochskaliert. Die Darstellung verwendet die invertierten Grauwerte, um einen größeren Kontrast zu bieten. Des Weiteren wird mit $\gamma = 0.1$ eine Gamma-Korrektur vorgenommen.

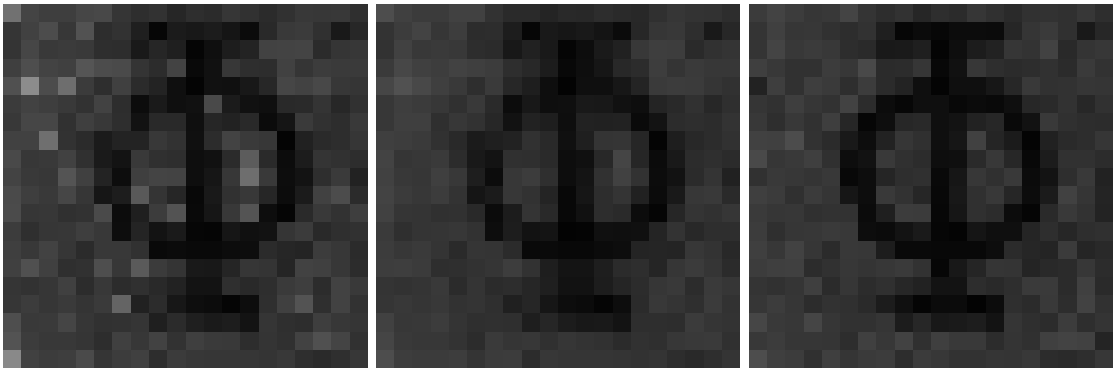
angestrebten Filterung vorhanden. Dies bedeutet, dass das Netz (zumindest innerhalb von 1000 Iterationen) nicht in der Lage ist, die Projektionen des Trainings vollständig an das rauschfreie Pendant anzupassen. Dies zeigt zunächst, dass das Netz in seiner Architektur nicht gänzlich die Streuverteilung aller 100 Trainingsprojektionen (Datensatz III) speichern kann. Hierdurch ist eine Überanpassung bereits erschwert. Allerdings ist nicht feststellbar, ob das Netz überhaupt genügend Informationen zur Bewertung aufnehmen kann.
Im Folgenden wird das neuronale Netz stets über 1000 Iterationen trainiert, um das volle Potential auszuschöpfen.

8.1.3 Datensatz I - Trainingsdaten

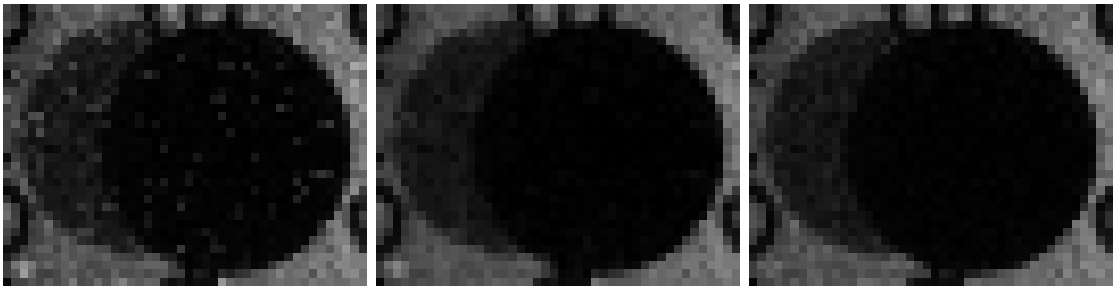
Das Training des neuronalen Netzes mit Datensatz I wird anhand eines Trainings über 1000 Iterationen bewertet.
Abbildung 8.7 zeigt eine Projektion und zwei verschiedene Ausschnitte des Trainingsdatensatzes unbearbeitet, durch das Netz nach 1000 Iterationen gefiltert und aus der Simulation optimiert (Ground Truth). In der unbearbeiteten Projektion 8.7a ist ein hoher Rauschanteil vorhanden. 8.7c zeigt dagegen nur noch das statistische (hochfrequente) Rauschen. Im Detailausschnitt 8.7i ist gut zu erkennen, dass in den Bereichen hoher Absorption keinerlei Bildrauschen vorhanden ist. Das Raschen in der dunklen (stärker absorbierenden) Kugel ist komplett entfernt, während es in dem helleren Bereich (linke Halbkugel) lediglich deutlich reduziert ist. Dieses Rauschen wird nicht der Streustrahlung zugeordnet, sondern kann der Statistik der Röntgenstrahlung zugeschrieben werden.
Die durch das Netz gefilterte Projektion weist ein geringeres Rauschen im Vergleich zur Ground Truth auf. Da die Projektion auch deutlich *weicher* erscheint, muss untersucht werden, ob damit auch ein Informationsverlust einhergeht. Bei Vergleich zwischen einem Ausschnitt des Originals (8.7g) und der Filterung (8.7h) fällt eine deutliche Rauschreduktion durch die Filterung auf. Die Materialgrenzen zwischen der rechten und linken Kugel sowie der linken Kugel und dem Hintergrund sind besser aufgelöst. Die Auflösung der Φ's in 8.7d und 8.7e dagegen ist vom Original zur Filterung nicht eindeutig verbessert. Ein zusätzlicher Vergleich zeigt, dass gerade im oberen und im unteren Bereich des Buchstaben die Ungenauigkeit in der horizontalen Achse erhöht ist. Hier wurden die Strukturen des unbearbeiteten Bildes verbreitert.
Es bleibt somit festzustellen, dass die Objektgröße eine wichtige Rolle in der Optimierbarkeit durch das entwickelte Netz darstellt. Das das Φ verschlechtert wird, spricht dafür, dass es von dem Netz nicht von Bildrauschen unterschieden werden kann und insofern hier eine Reduzierung der Auflösung ensteht.
In der Streumatrix in 8.8a ist die gesamte Phantomstruktur zu erkennen. Des Weiteren ist die erkannte Streustrahlung hochfrequent und in nahezu jedem Pixel sind Werte >0, was die Reduktion des statistischen Rauschens in Abbildung 8.7b erklärt. In der angestrebten Streuverteilung 8.8b sind deutlich gröbere, alse niederfrequente Strukturen vorhanden, welche keinerlei Objektionformationen erkennen lassen. Des Weiteren treten absolut größere Werte auf.
Es wird deutlich, dass durch die Objektinformation in der durch das Netz gefilterten Streustrahlung die Auflösung kleiner Strukturen verringert wird. Die Untersuchung des zweiten Datensatzes in Abschnitt 8.1.4 wird zeigen, ob das in der geringen Trainingsdatenmenge begründet liegt oder ein allgemeines Problem darstellt.



(a) Original **(b)** 1000 Iterationen **(c)** Ground Truth

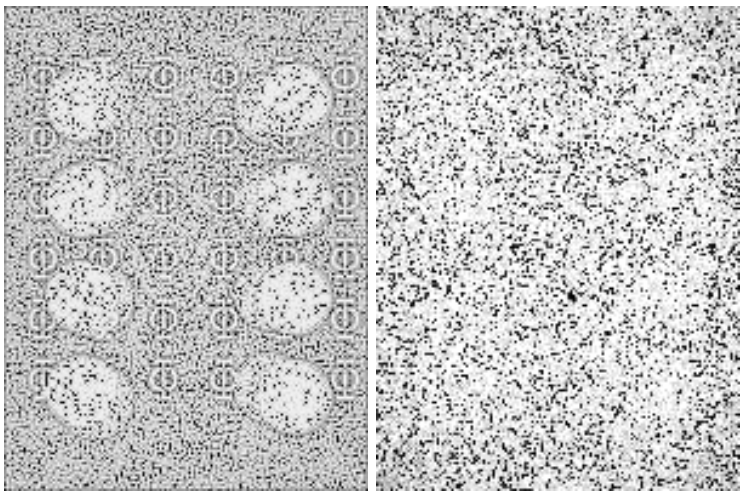


(d) Original **(e)** 1000 Iterationen **(f)** Ground Truth



(g) Original **(h)** 1000 Iterationen **(i)** Ground Truth

Abbildung 8.7 In den Abbildungen (a), (b) und (c) ist eine Projektion des Datensatzes I unbearbeitet, durch das Netz gefiltert und aus Simulation bestimmt, gezeigt. (d), (e) und (f) zeigen je den gleichen Ausschnitt dieser drei Projektionen. Selbiges gilt für (g), (h) und (i).



(a) 1000 Iterationen **(b)** Ground Truth

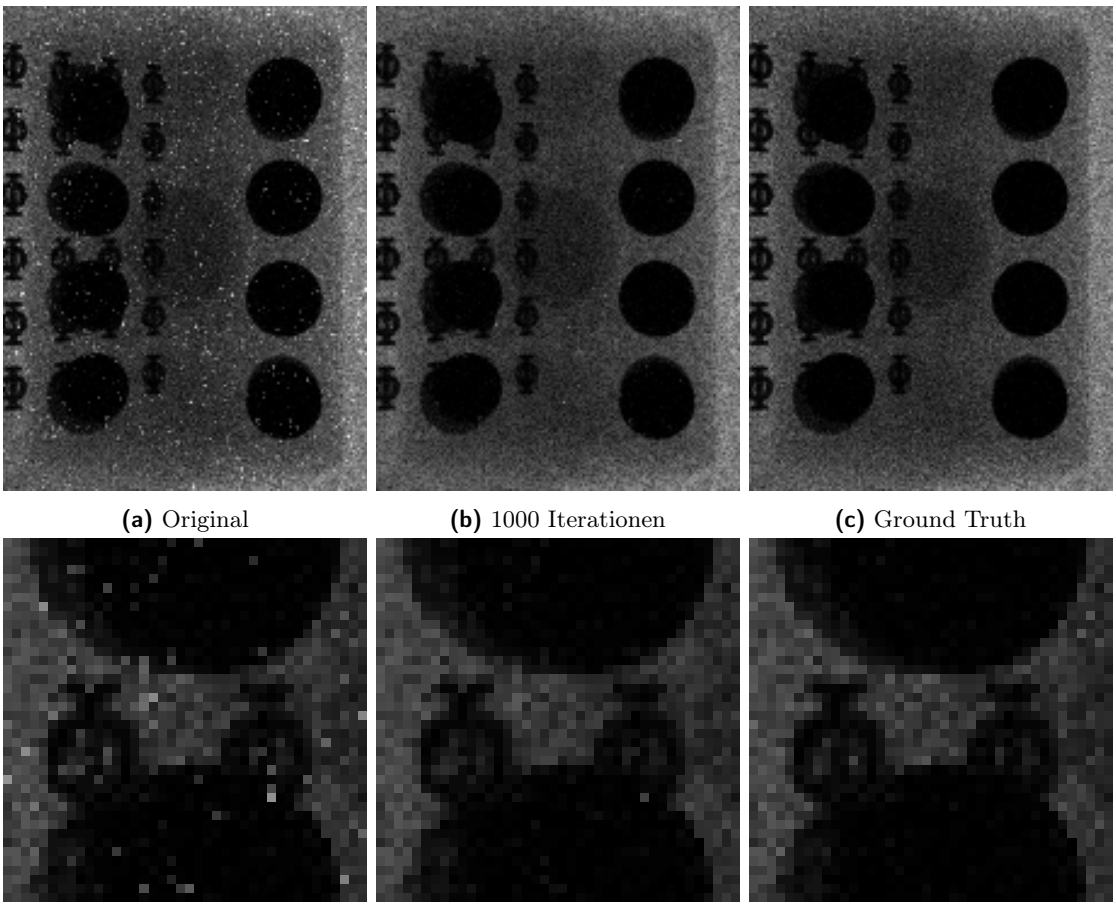
Abbildung 8.8 Zu sehen sind die Streumatrizen, welche sich aus dem neuronalen Netz (a) und der Simulation (b) ergeben (Datensatz I). Beide Bilder sind zum Maximum der Ground Truth (b) skaliert. Die Darstellung verwendet die invertierten Grauwerte, um einen größeren Kontrast zu bieten. Des Weiteren wird mit $\gamma = 0.1$ eine Gamma-Korrektur vorgenommen.

8.1.4 Datensatz II - Trainingsdaten

Das Training mit Datensatz II zeigt vergleichbare Ergebnisse zum Training mit Datensatz I im vorangegangenen Abschnitt 8.1.3. Die Abbildungen in 8.9 zeigen eine deutliche Reduktion der Streustrahlung und somit des niedrigfrequenten Bildrauschens zwischen dem Original und der gefilterten sowie der simulierten Projektion in 8.9a, 8.9b und 8.9c. Die gefilterte Projektion wirkt allerdings nicht viel *weicher* als die simulierte. Das heißt, dass das niedrigfrequente Rauschen nicht in gleicher Form reduziert wird, wie das bei Datensatz I der Fall ist.
Die Detailausschnitte 8.9d, 8.9e und 8.9f zeigen, dass in den großen Strukturen das Rauschen reduziert und die Materialgrenzen erkennbar sind. Die feinen Strukturen sind ähnlich wie bei Datensatz I in ihrer Auflösung nicht verbessert. Auch hier findet eher eine Verschlechterung der Auflösung als eine Verbesserung statt.
Die beiden in Abbildung 8.10 dargestellten Streumatrizen ähneln sich zunächst, was die Frequenz und Größenordnung der Streustrahlung anbelangt. Die maximale Intensität ist in der Ground Truth-Projektion höher und somit ist auch mehr Dynamik vorhanden. In der Projektion des neuronalen Netzses wird ein Anteil der Objektinformation herausgefiltert. Es sind die Kugeln zu erkennen und am Linken Rand zeigen sich Strukturen, die auf die Φ's zurückzuführen sind. Vergleicht man die gefilterten Strukturen mit der Filterung durch Datensatz I in Abbildung 8.8a, erreicht Datensatz II ein deutlich besseres Ergebnis.

8.1.5 Datensatz IV - Trainingsdaten

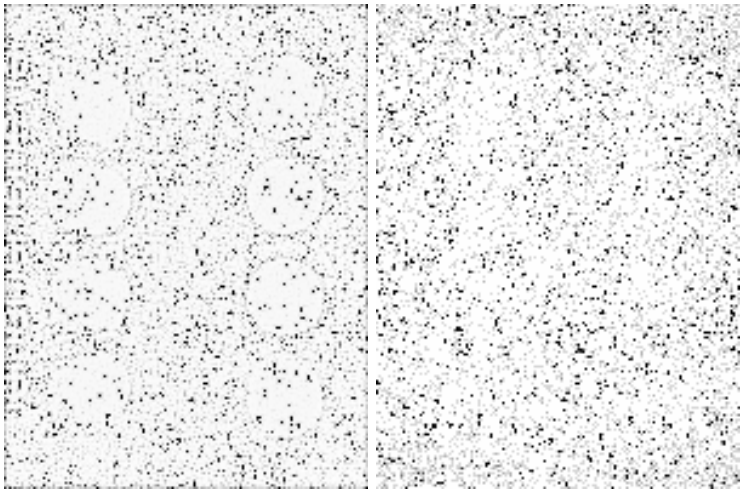
Datensatz IV unterscheidet sich stark von den übrigen Datensätzen. Hier ist das Phantom invertiert, das heißt der Zylinder besteht aus Aluminium und die enthaltenen Kugeln sind mit Luft gefüllt. Gleiches gilt für den Quader, in welchem sich die Φ's befinden (vgl. Abschnitt 5.4.1).
Abbildung 8.11 zeigt eine Projektion des Phantoms, in dem sich der Quader parallel zum Detektor, zwischen Detektor und Phantom, befindet. Zwischen der ungefilterten (8.11a) und der



(a) Original **(b)** 1000 Iterationen **(c)** Ground Truth

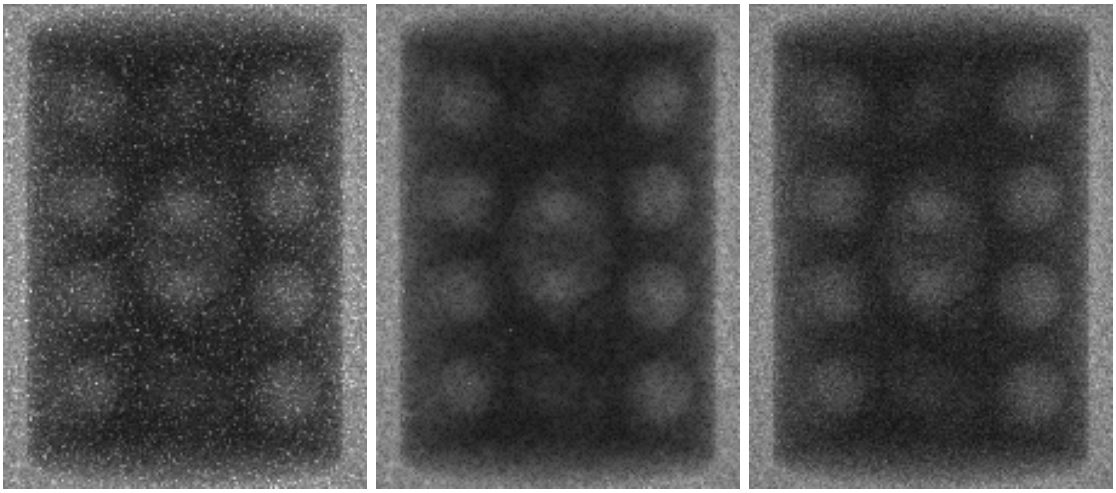
(d) Original **(e)** 1000 Iterationen **(f)** Ground Truth

Abbildung 8.9 In den Abbildungen (a), (b) und (c) ist eine Projektion des Datensatzes II unbearbeitet, durch das Netz gefiltert und aus Simulation bestimmt, gezeigt. (d), (e) und (f) zeigen je den gleichen Ausschnitt dieser drei Projektionen.

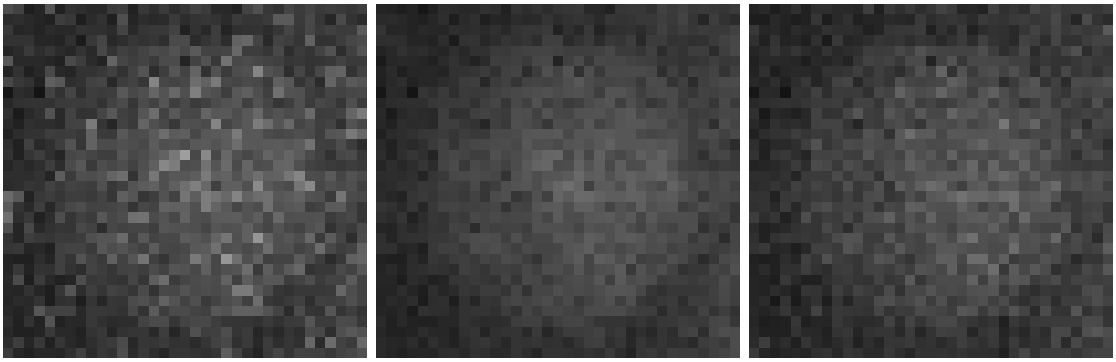


(a) 1000 Iterationen **(b)** Ground Truth

Abbildung 8.10 Zu sehen sind die Streumatrizen, welche sich aus dem neuronalen Netz (a) und der Simulation (b) ergeben (Datensatz II). Beide Bilder sind zum Maximum der Ground Truth (b) skaliert. Die Darstellung verwendet die invertierten Grauwerte, um einen größeren Kontrast zu bieten. Des Weiteren wird mit $\gamma = 0.1$ eine Gamma-Korrektur vorgenommen.

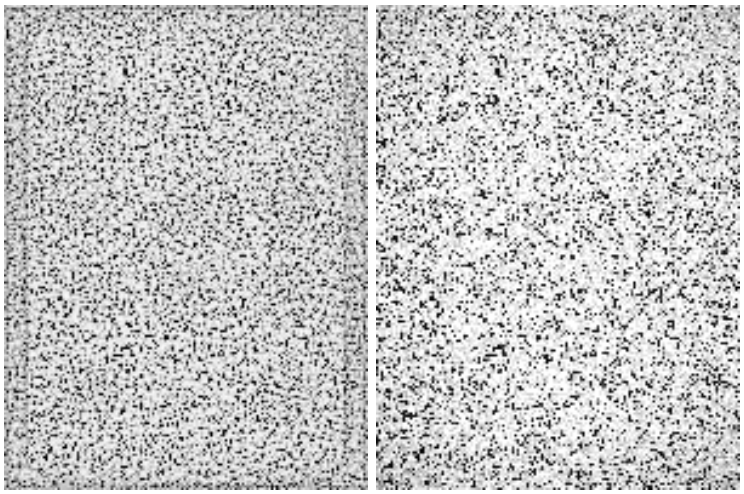


(a) Original **(b)** 1000 Iterationen **(c)** Ground Truth



(d) Original **(e)** 1000 Iterationen **(f)** Ground Truth

Abbildung 8.11 In den Abbildungen (a), (b) und (c) ist eine Projektion des Datensatzes IV unbearbeitet, durch das Netz gefiltert und aus Simulation bestimmt, gezeigt. (d), (e) und (f) zeigen je den gleichen Ausschnitt dieser drei Projektionen.



(a) 1000 Iterationen **(b)** Ground Truth

Abbildung 8.12 Zu sehen sind die Streumatrizen, welche sich aus dem neuronalen Netz (a) und der Simulation (b) ergeben (Datensatz IV). Beide Bilder sind zum Maximum der Ground Truth (b) skaliert. Die Darstellung verwendet die invertierten Grauwerte, um einen größeren Kontrast zu bieten. Des Weiteren wird mit $\gamma = 0.1$ eine Gamma-Korrektur vorgenommen.

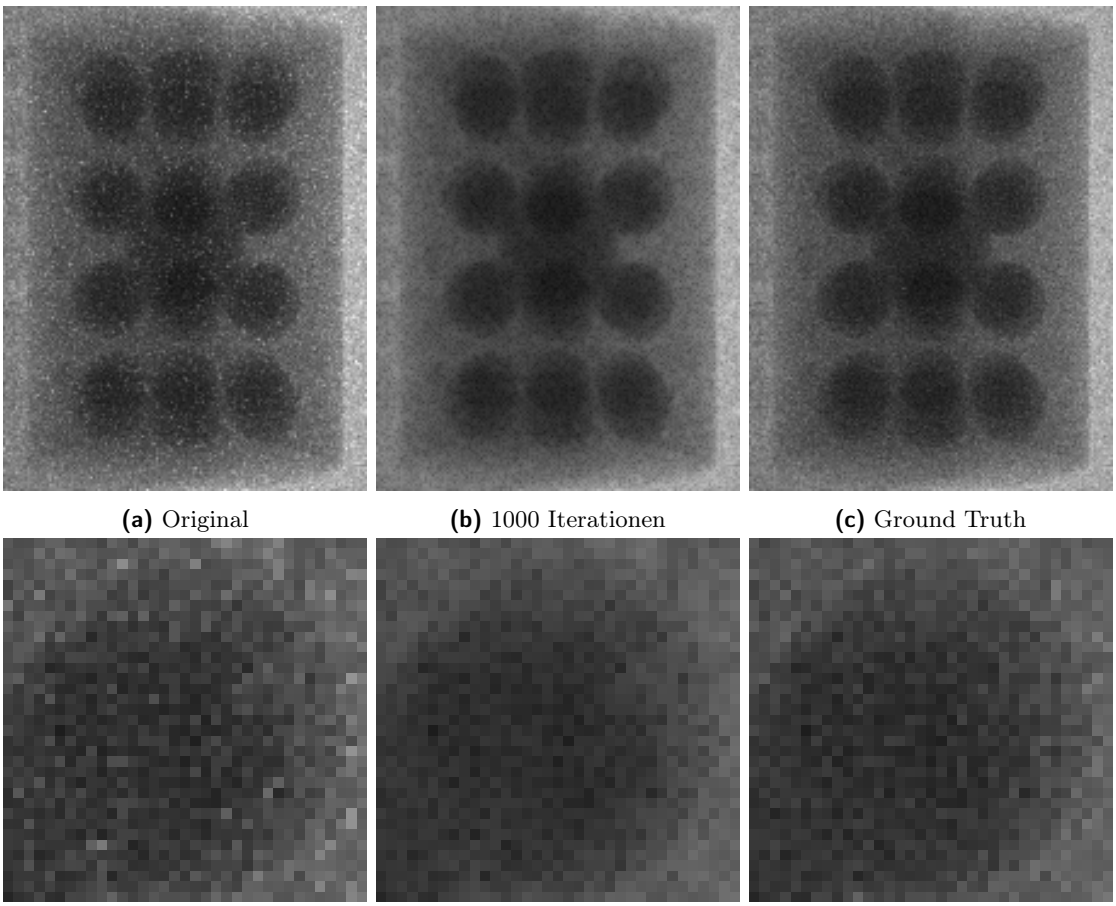
simulierten Projektion (8.11c) ist eine Reduktion der Streustrahlung erkennbar. Die gefilterte Projektion (8.11b) weist dagegen auch eine Reduktion des hochfrequenten Rauschens auf. Dies lässt das Bild deutlich *weicher* wirken.
Die Detailausschnitte zeigen die zweite Aushöhlung von unten am rechten Rand. In allen Projektionen ist die Trennung zwischen Hohlraum und Zylinder am oberen rechten Rand am besten. An den übrigen Rändern ist die Erkennung einer Materialgrenze deutlich erschwert. Hier zeigt sich, dass sowohl die gefilterte, als auch die simulierte Projektion die Auflösung verbessern können. In der unbearbeiteten Projektion 8.11d ist das Rauschen im Kantenbereich zu groß, um eine Randdefinition zu ermöglichen. Die Simulation in 8.11f und die Filterung in 8.11e erlauben ebenfalls keine gute Materialtrennung. Gerade die gefilterte Abbildung zeigt einen Grauwert-Verlauf, der die Definition einer Grenzfläche stark erschwert. Die Ground Truth-Projektion lässt durch das Rauschen ebenfalls keine bessere Materialtrennung zu.
Insofern kann die Filterung hier als Verbesserung interpretiert werden. Da die feinen Strukturen der Φ-Matrix ohnehin nicht aufgelöst werden können, funktioniert die Filterung zur besseren Auflösung der groben Strukturen der Hohlräume.
Die in Abbildung 8.12a gezeigte Ground Truth-Streustrahlung umfasst mehr Pixel und somit höhere Raumfrequenzen, als die aus der Simulation ermittelte Streustrahlung (8.12b). Dies deckt sich mit der starken Verringerung des Bildrauschens in 8.11b. Die Dynamik im durch das Netz ermittelten Streubild ist geringer als im Ground Truth-Streubild. Es werden durch das neuronale Netz somit höhere Rauschfrequenzen, die nicht der Streustrahlung entsprechen, herausgefiltert. Es ist der Umfang des Zylinders zu erkennen. Die Aushöhlungen dagegen sind nicht zu verorten. Die verlorene Objektinformation ist also innerhalb des Zylinders gering.

8.2 Anwenden des Netzes auf Testdaten

Die Testdaten enststammen dem gleichen Datensatz wie die Trainingsdaten. Sie stellen hier lediglich andere Projektionen des gleichen Phantoms dar. Die Testdaten dienen dazu, die Anwendbarkeit des Netzes zu prüfen, bevor es abschließend trainiert wird. So wird das Netz bewertet, um daraufhin die Hyperparameter anzupassen. Das Netz wird also nicht direkt mit den Testdaten trainiert, ein indirekter Einfluss auf das Netz besteht aber. Insofern taugen die Testdaten nicht als Validierungsdaten, um abschließend die Güte des Netzes zu bewerten. Die hohe Ähnlichkeit zwischen Trainings- und Testdaten wie in diesem Falle ist nicht optimal, falls eine breitere Anwendbarkeit (wie sie durch die Validierungsdaten geprüft wird) gewünscht ist.
Aufgrund der hohen Ähnlichkeit lassen sich über die Testadaten die gleichen Aussagen treffen, wie im vorangegangen Abschnitt über die Trainingsdaten. Aus diesesm Grund sind die Projektionen und Details der Testdaten der Datensätze (I, II und IV) in den Anhang (siehe A.1.2) verschoben. Im Folgenden wird anhand von Datensatz III eine Projektion des Testdatensatzes vorgestellt.

8.2.1 Datensatz III - Testdaten

Nachdem das Netz mit den Trainingsdaten über 1000 Zyklen trainiert wurde, wird das Netz auf die Testdaten angewendet. Abbildung 8.13 zeigt die Anwendung auf eine Testprojektion des Datensatzes III. 8.13a zeigt das ungefilterte Bild. Es ist grobes Bildrauschen im Vergleich zu den

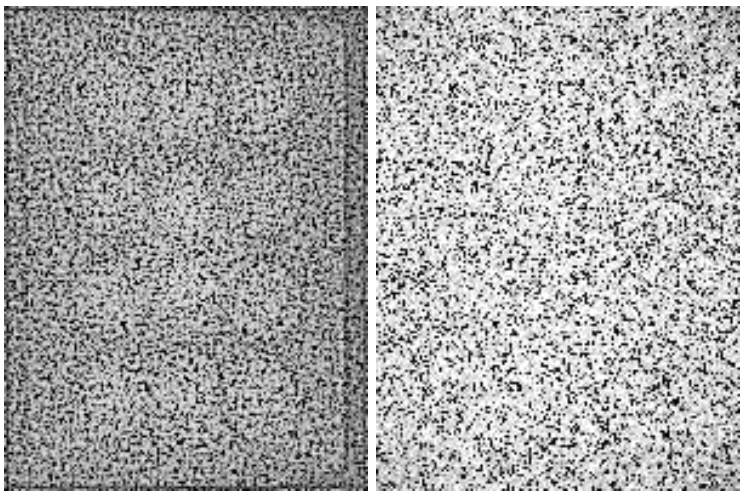


(a) Original **(b)** 1000 Iterationen **(c)** Ground Truth

(d) Original **(e)** 1000 Iterationen **(f)** Ground Truth

Abbildung 8.13 (a), (b) und (c) zeigen die ungefilterte, die durch das Netz bestimmte (1000 Iterationen) und die simulierte Ground Truth-Projektion aus dem Testdatensatz III. (d), (e) und (f) zeigen jeweils einen Ausschnitt um die zweite Kugel von oben, rechts.

Abbildungen 8.13b und 8.13c zu erkennen, welche das gefilterte und das simulierte Bild (Ground Truth) zeigen. Zunächst wirkt das gefilterte Bild am rauschärmsten. Es ist deutlich rauschärmer als die Ground Truth-Projektion.
Die Abbildungen 8.13d, 8.13e und 8.13f zeigen jeweils eine Kugel aus der ungefilterten, der gefilterten und der simulierten Projektion. In dem Ausschnitt sind zwei Kugeln voreinander durchleuchtet. Eine größer wirkende Kugel befand sich näher an der Röntgenquelle und eine weniger vergrößerte Kugel weiter von der Quelle entfernt. Die stärker vergrößerte Kugel ragt oben rechts über die andere Kugel hinaus. Ein Qualitätskriterium ist, ob sich die Bereiche *keine Kugel* , *eine Kugel* und *zwei Kugeln* im Strahlengang voneinander trennen lassen. Eine solche Unterscheidung ist in allen drei Projektionen nicht exakt vorzunehmen. Die ungefilterte Abbildung lässt aufgrund des großen Schwankungen der Pixelwerte lediglich im Bereich oben links eine klare Unterscheidung *zwei Kugeln* gegen *keine Kugel* zu. Die Trennung *eine Kugel* gegen *zwei Kugeln* ist in der gefilterten und simulierten Abbildung möglich, allerdings lässt sich hier keine klare Kante definieren. Eine Unterscheidung *eine Kugel* zu *keine Kugel* ist noch schwerer definierbar. Die gefilterte Projektion weist weniger hochfrequentes Bildrauschen auf, als die Ground Truth-Projektion.
Abbildung 8.14 zeigt das durch das Netz ermittelte Rauschbild und das Ground Truth-Streubild. Die zuvor erarbeitete Charakteristik bestätigt sich. 8.14a zeigt deutlich mehr angepasste Pixel



(a) 1000 Iterationen **(b)** Ground Truth

Abbildung 8.14 Zu sehen sind die Streumatrizen, welche sich aus dem neuronalen Netz (a) und der Simulation (b) ergeben (Testdatensatz III). Beide Bilder sind zum Maximum der Ground Truth (b) skaliert. Die Darstellung verwendet die invertierten Grauwerte, um einen größeren Kontrast zu bieten. Des Weiteren wird mit $\gamma = 0.1$ eine Gamma-Korrektur vorgenommen.

und somit eine deutlich höhere Rauschfrequenz, die aus dem Bild entfernt wird. 8.14b weist dagegen weniger Pixel mit Streustrahlung auf. Im gefilterten Bild ist außerdem die Kontur des Zylinders sowie der Kugeln zu erkennen. Im rechten Bildbereich sind diese ausgeprägter. Im linken Bildbereich ist die Φ-Matrix im Strahlengang. Es wird somit in den Bereichen ohne die Φ-Matrix mehr Bildinformation durch die Filterung entfernt.

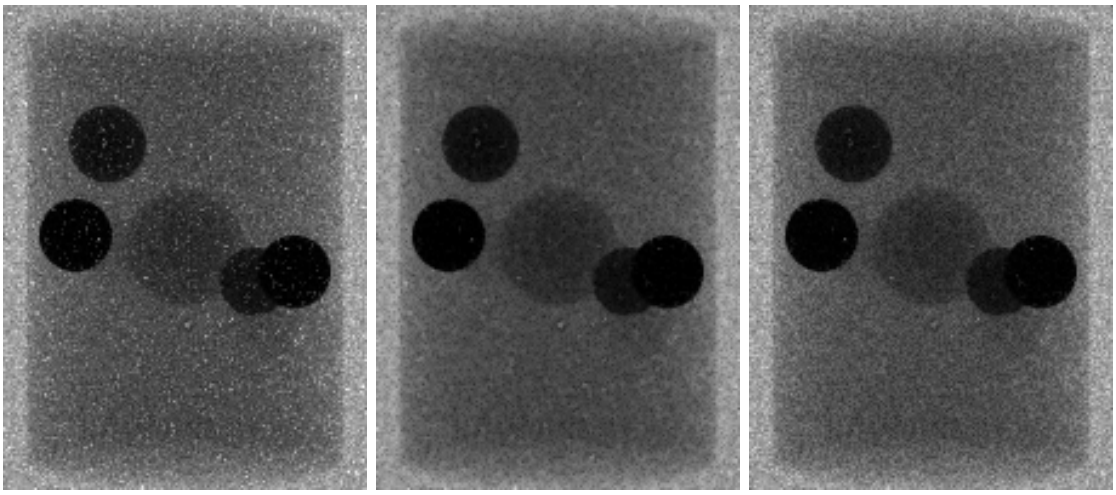
8.3 Anwenden des Netzes auf Validierungsdaten

Im Folgenden wird Streustrahlerkennung auf zwei Validierungsphantome angewendet. In Abschnitt 5.4 sind diese Phantome in den Abbildungen 5.8b und 5.8d zu sehen. Die Phantome aus den Abbildungen 5.8c und 5.8e sind im Anhang unter Abschnitt A.1.3 mit Detailausschnitten dargestellt.

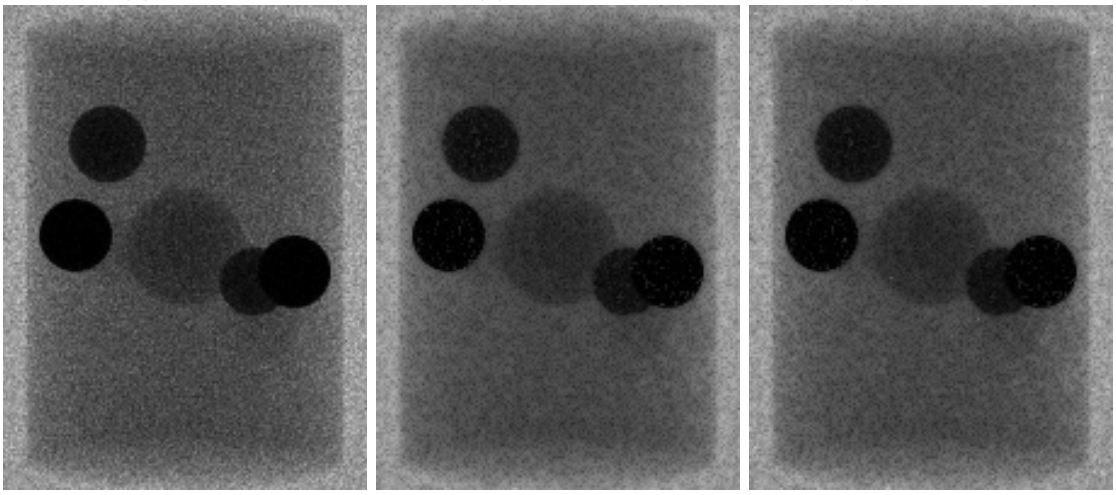
8.3.1 Validierungsphantom (b)

Abbildung 8.15 zeigt, dass das Validierungsphantom unterschiedlich gefiltert wird, je nachdem mit welchem Datensatz das neuronale Netz trainiert wurde. Die Projektion 8.15b ergibt sich aus der Filterung nach Training mit Datensatz I. Das resultierende Bild wirkt sehr weich, beeinhaltet aber noch einige große Artefakte. Das mit Datensatz II trainierte System erzeugt kein weiches Bild (vgl. 8.15c). Das hochfrequente Rauschen bleibt, wie im simulierten Bild erhalten. Auch hier sind noch Teile der Streustrahlung als Artefakte zu erkennen. Datensatz III führt zu einem Netz, was nahezu alle ausgeprägten Artefakte unterdrückt (vgl. 8.15e). Im Vergleich zur simulierten Projektion wirkt auch dieses Bild deutlich *weicher*, das heißt, es wird auch das statistische Rauschen reduziert. Der vierte Datensatz in 8.15f zeigt eine sehr ähnliche Filterung zum dritten Datensatz. Allerdings werden die groben Artefakte nicht so stark herausgefiltert.

In Abbildung 8.16 wird ein Materialübergang sowie ein durch Streustrahlung entstandenes Bild-



(a) Original **(b)** Datensatz I **(c)** Datensatz II

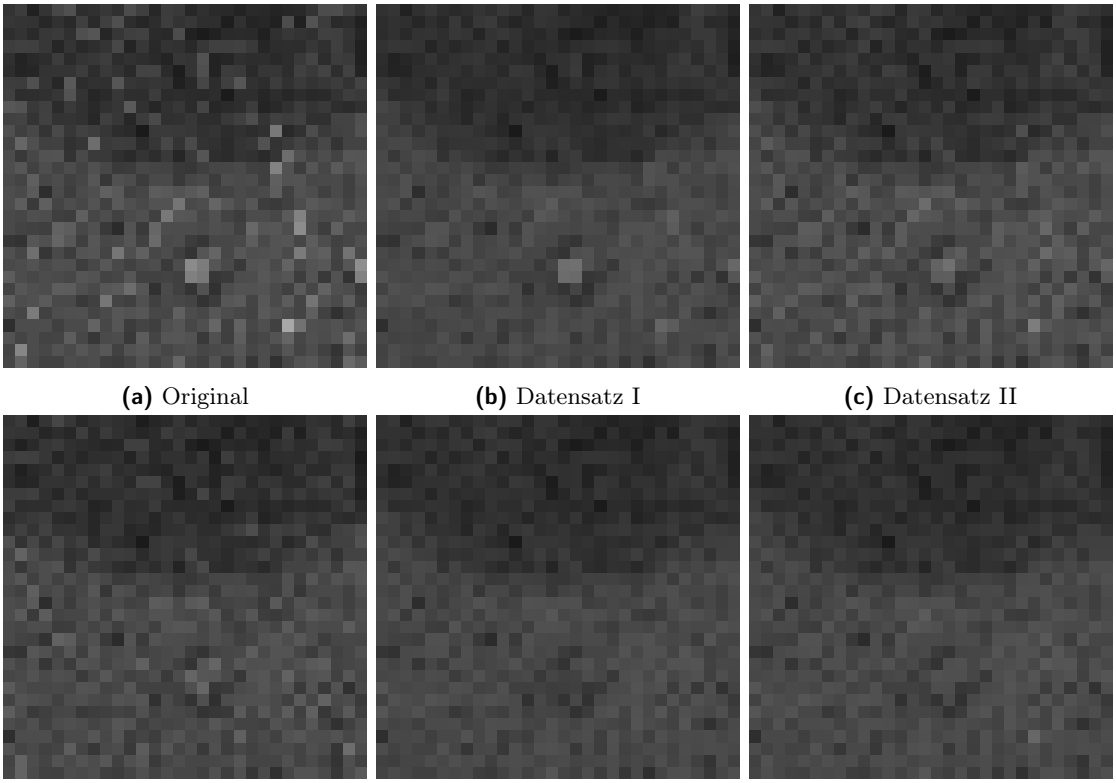


(d) Ground Truth **(e)** Datensatz III **(f)** Datensatz IV

Abbildung 8.15 Es wird die Projektion des Validierungsphantoms b im Original in Abbildung (a), nach den Datensätzen I, II, III und IV gefiltert in (b), (c), (e) und (f) und als Ground Truth in Abbildung (d) dargestellt.

artefakt betrachtet. Die unkorrigierte Abbildung 8.16a zeigt, dass der Materialübergang durch das Bildrauschen nicht klar erkennbar ist. Des Weiteren sind im unteren mittleren Teil des Bildes vier benachbarte Pixel besonders hell. Dies kannals Bildfehler durch Streustrahlung gewertet werden, da das Artefakt in der Abbildung 8.16d, welche die Ground Truth des Ausschnittes zeigt, nicht in dieser Form erkennbar ist. Auch im simulierten Bild ist dort eine markante Struktur zu sehen. Da allerdings der Bildfilter auf Basis der Weglänge nur eine Abschätzung macht, kann man das Artefakt dennoch auf Streustrahlung zurückführen.
Das Detail in Abbildung 8.16b zeigt, dass das Artefakt durch das mit Datensatz I trainierte Netz nicht korrigiert wurde. Weitere Pixel, die im unbearbeiteten Bild deutlich zu hohe Werte aufweisen, werden reduziert und heben sich nur noch minimal vom Hintergrund ab. Das statistische Rauschen ist im Vergleich zur simulierten Projektion deutlich geringer, was dem Bild einen weichen Eindruck gibt. Dies deckt sich mit dem Eindruck des Gesamtbildes. Auch Datensatz II (Abbildung 8.16c) kann das Artefakt nicht vollständig entfernen. Durch das höhere Bildrauschen ist es aber kaum erkennbar. Ebenso wie die Materialgrenze gleicht die Erkennbarkeit der in der Ground Truth-Projektion 8.16d.
Datensatz III erzeugt ein Netz, was das Artefakt vollständig aus dem Bild (8.16e) entfernt. Auch das Bildrauschen ist hier stärker reduziert als im angestrebten Bild. Die Materialgrenze in der Mitte des Ausschnitts ist ähnlich gut zu erkennen, wie das im simulierten Bild der Fall ist. Das Detail in Abbildung 8.16f zeigt nur minimale Unterschiede zur Filterung mit Datensatz III. Auch hier ist der Artefakt fast vollständig gefiltert.

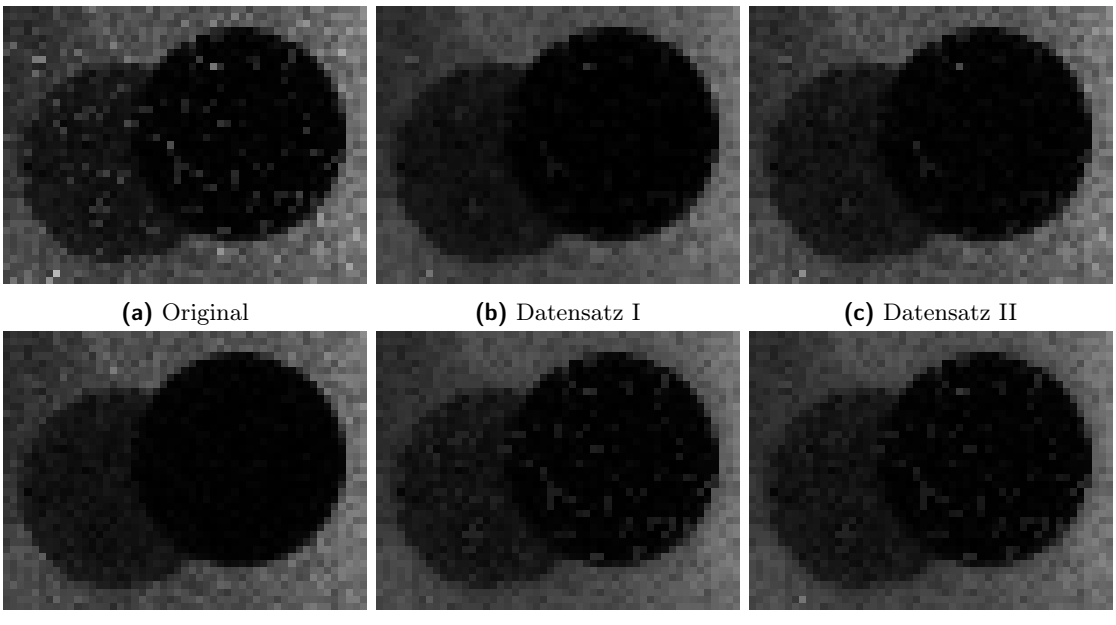
Als zweites Detail wird in Abbildung 8.17 der Materialübergang zwischen zwei Kugeln unterschiedlicher Materialien betrachtet. Datensatz I zeigt in Abbildung 8.17b ähnliche Eigenschaften zur Gesamtansicht und dem ersten Detail. Das Rauschen innerhalb der Kugeln ist sehr stark reduziert. Im linken Bildteil, oben und unten sowie in der dunklen Kugel oben in der Mitte ist zu sehen, das kumulierte Streustrahlung nicht entfernt werden kann, sondern lediglich reduziert wird. Der Bereich, in welchem keine Kugel im Strahlengang ist, ist ebenfalls stark rauschreduziert. Hier und in der linken Kugel ist das Rauschen deutlich stärker reduziert, als dies bei der simulierten Projektion (Ground Truth) der Fall ist. Die Reduktion des Rauschens in der linken Kugel führt zu einer allgemeinen Absenkung der Helligkeit in diesem Bereich. Durch den verminderten Unterschied der Grauwerte wirken die Materialien der Kugeln ähnlicher. In Datensatz II 8.17c lassen sich die beiden Kugeln deutlicher voneinander trennen. Die am stärksten durch die Streustrahlung hervorgehobenen Pixel sind auch im gefilterten Bild noch zu erkennen. Die Kugeln sind diskret vom Hintergrund zu unterscheiden. Das Bildrauschen gleicht dem der simulierten Projektion in 8.16d. Abbildung 8.17e zeigt die Filterung durch Datensatz III. Hier sind das Rauschen innerhalb der Kugeln und die Streustrahlung nur leicht verringert. Das Rauschen im Bereich ohne Kugeln ist dagegen im Vergleich zur simulierten Projektion reduziert. Bei der Filterung nach Datensatz IV gleicht das Bild stark der Filterung durch Datensatz III. Außerhalb der Kugeln ist die Streustrahlung etwas weniger reduziert. Das Rauschen innerhalb der Kugeln ist mit Datensatz III vergleichbar.



(a) Original **(b)** Datensatz I **(c)** Datensatz II

(d) Ground Truth **(e)** Datensatz III **(f)** Datensatz IV

Abbildung 8.16 Es wird ein Ausschnitt des Validierungsphantoms b im Original in Abbildung (a), nach den Datensätzen I, II, III und IV gefiltert in (b), (c), (e) und (f) und als Ground Truth in Abbildung (d) dargestellt.



(a) Original **(b)** Datensatz I **(c)** Datensatz II

(d) Ground Truth **(e)** Datensatz III **(f)** Datensatz IV

Abbildung 8.17 Es wird ein Ausschnitt des Validierungsphantoms b im Original in Abbildung (a), nach den Datensätzen I, II, III und IV gefiltert in (b), (c), (e) und (f) und als Ground Truth in Abbildung (d) dargestellt.

8.3.2 Validierungsphantom (d)

Als zweites wird das Validierungsphantom (d) mit den unterschiedlich trainierten Netzen gefiltert. Abbildung 8.18 zeigt die gefilterten Projektionen für alle Datensätze sowie die unbearbeitete und die nach den Simulationsdaten gefilterte Projektion (Ground Truth). Es ist erkennbar, dass jedes Netz die Streustrahlung reduziert. Unterschiede liegen hier auf den ersten Blick in der Schärfe der Bilder und in der Reduktion der Streustrahlung innerhalb der Kugeln. Die Schärfe wird hauptsächlich durch das Rauschen bestimmt. Sind die Strukturen klar erkennbar und umfasst das Rauschen keine pixelübergreifenden Strukturen, wirkt das Bild scharf trotz des Rauschens. Tritt das Rauschen als ein Effekt auf, der pixelübergreifende Strukturen erschafft bzw. beinflussen sich die benachbarten Pixelwerte objektunabhängig, so wirkt das Bild unscharf und verrauscht, weil die Grauwerte mehrerer Pixel nebeneinander ähnliche Werte haben bzw. *ineinander übergehen*. Hierdurch werden auch die Objektränder *verwischt*. Sowohl in der ungefilterten, als auch in der nach den Simulationsdaten gefilterten Projektion ist Bildrauschen auf Pixelebene vorhanden. Dieses Rauschen ist der Statistik zuzusprechen und entspricht nicht der Streustrahlung, die hier gefiltert werden soll.

In Abbildung 8.18b ist die Projektion durch das mit Datensatz I trainierte Netz gefiltert. Die entstehende Projektion ist sehr weich. 8.18c (Datensatz II) ist dagegen schärfer und die Objekte sind definierter zu erkennen. Die Filterung mit Datensatz III (8.18e) erreicht eine ähnliche Schärfe wie die Filterung durch Datensatz II. Hier bleibt in den Kugeln deutlich mehr Streustrahlung vorhanden. Datensatz IV filtert sehr ähnlich zu Datensatz III. Bei Betrachtung der gesamten Projektion sind keine Unterschiede zu erkennen.

Das Detail in Abbildung 8.19a zeigt die Konturen eines Φ's, einige Bildartefakte und eine Materialgrenze. Die Artefakte umfassen sowohl markante Streustrahlung in einzelnen Pixeln (Mitte oben), als auch größere Strukturen (Mitte), welche auch die Auflösung des Objekts (Φ) (links Mitte) verschlechtern. Anhand dieser Merkmale wird erörtert, inwiefern die unterschiedlich trainierten Netze diese Bildfehler erkennen und korrigieren können.

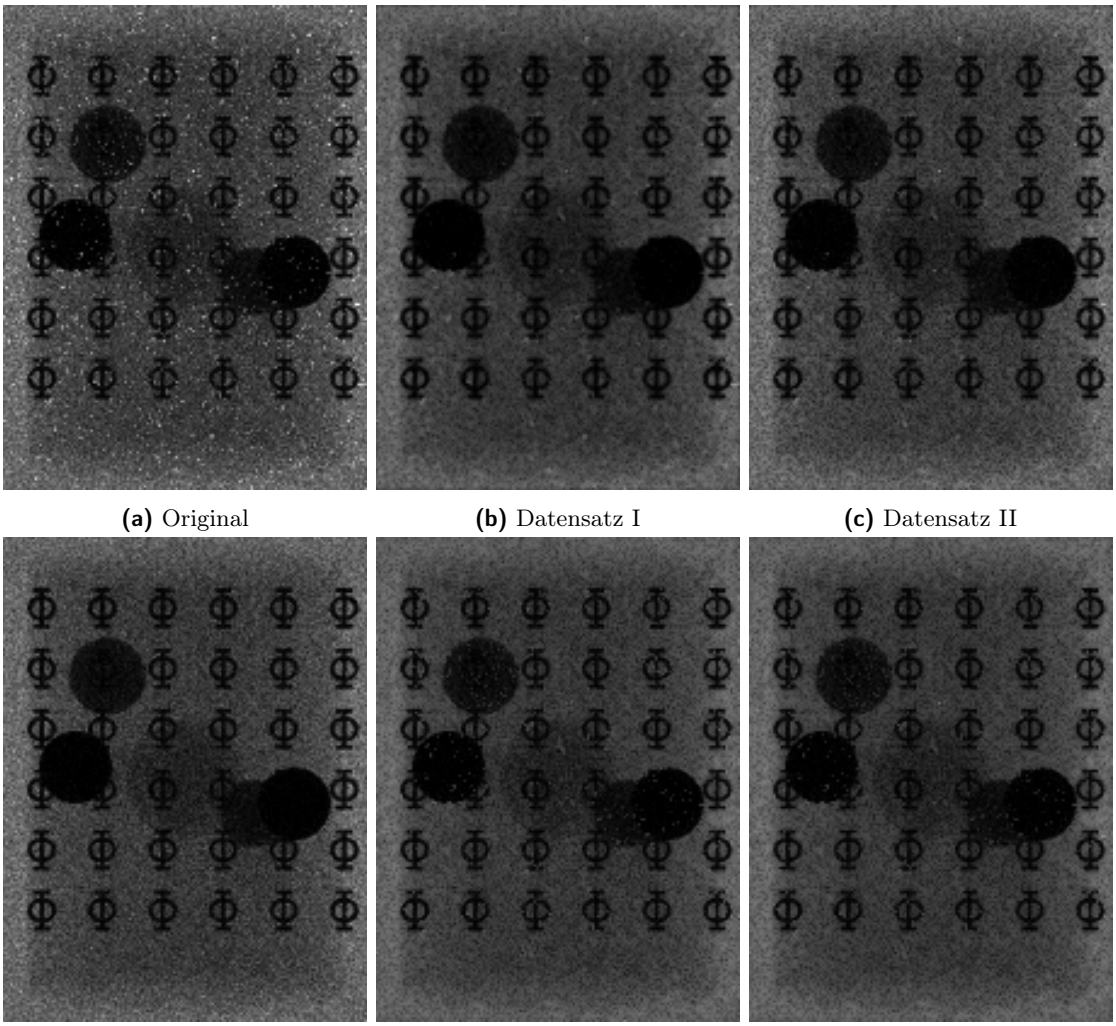
8.19b zeigt die Filterung durch Datensatz I. Hier sind die meisten Artefakte noch zu erkennen, ihre Intensität und Ausdehnung ist aber verringert. Die Objekte sind schlechter aufgelöst als in der ursprünglichen Projektion. Die Objektkanten wurden verbreitert. Die Materialgrenze ist zu erkennen. Das Bildrauschen ist gering.

In Abbildung 8.19c ist die Filterung durch Datensatz II zu sehen. Die Artefakte werden in ähnlicher Weise wie bei Datensatz I unterdrückt. Hier ist das Rauschen nicht in gleicher Stärke verringert und die Strukturen sind nicht in gleicher Weise vergrößert (Unschärfe). Das Rauschen ist also größer und dennoch sind die Strukturen besser zu erkennen.

Datensatz III filtert die Artefakte anders (8.19e). Sie werden nicht in ihrer Struktur verändert, sondern lediglich in ihrer Intensität verringert. Das Bildrauschen gleicht der angestrebten Projektion 8.19d. Die Objektauflösung ist im Vergleich zum Original weder verschlechtert noch verbessert. Die Materialgrenze ist ähnlich gut wie im angestrebten Bild zu erkennen.

Die Filterung durch Datensatz IV weist keine signifikanten Unterschiede zur Filterung mit Datensatz III auf.

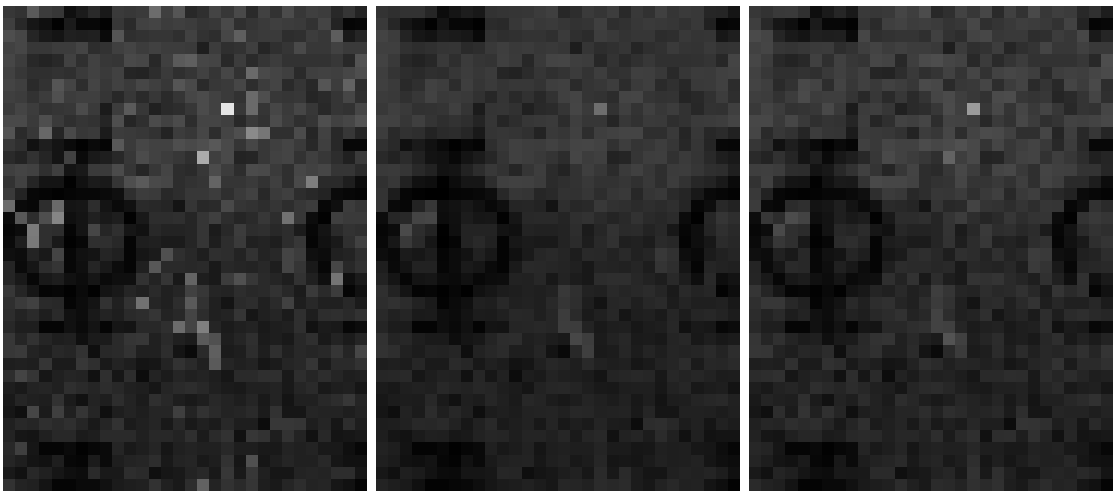
Abbildung 8.20 zeigt weitere Detailausschnitte. Hier liegt der Fokus auf den groben Strukturen (den Kugeln) und ihren Materialgrenzen. Das Original 8.20a weist Streustrahlung auf, welche die Erkennung von Materialgrenzen (Übergang rechte Kugel-linke Kugel und rechte Kante-rechte



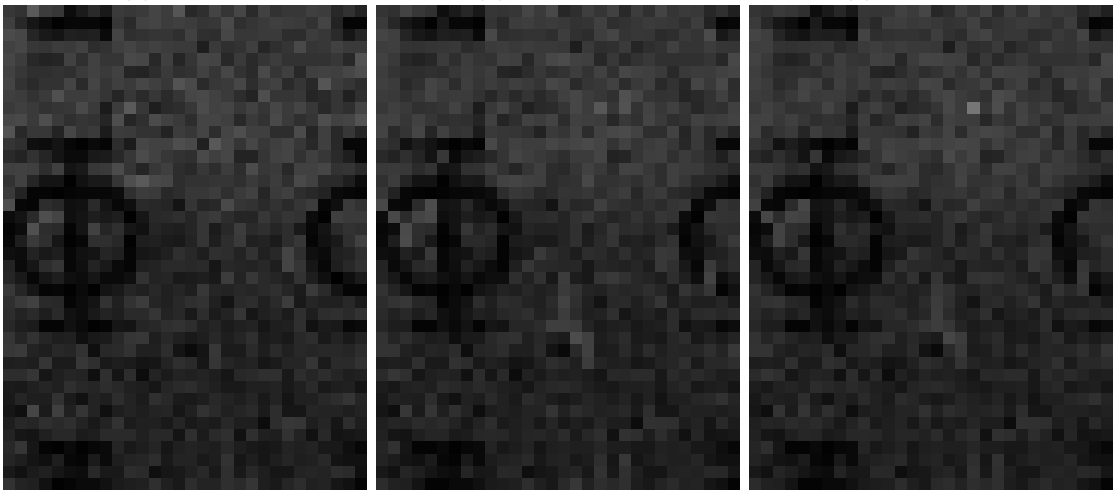
(a) Original **(b)** Datensatz I **(c)** Datensatz II

(d) Ground Truth **(e)** Datensatz III **(f)** Datensatz IV

Abbildung 8.18 Es wird das Validierungsphantoms d im Original in Abbildung (a), nach den Datensätzen I, II, III und IV gefiltert in (b), (c), (e) und (f) und als Ground Truth in Abbildung (d) dargestellt.



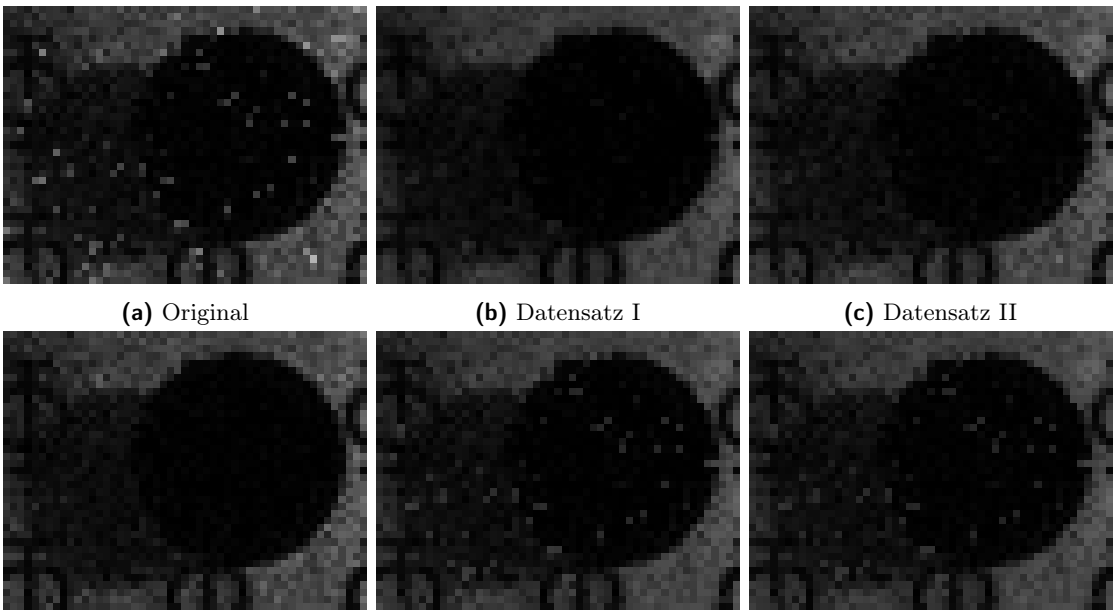
(a) Original **(b)** Datensatz I **(c)** Datensatz II



(d) Ground Truth **(e)** Datensatz III **(f)** Datensatz IV

Abbildung 8.19 Es wird ein Ausschnitt des Validierungsphantoms d im Original in Abbildung (a), nach den Datensätzen I, II, III und IV gefiltert in (b), (c), (e) und (f) und als Ground Truth in Abbildung (d) dargestellt.

Kugel) erschwert. Des Weiteren ist am oberen linken Rand des Bildes noch eine Detailstruktur (Φ) deren Auflösung erschwert ist.
Die Filterung durch Datensatz I (8.20b) entfernt die Streustrahlung und reduziert das Bildrauschen darüber hinaus. Die Materialgrenze zwischen den Kugeln ist nicht zu erkennen. Die Kugeln scheinen aus einem Material sehr ähnlicher Absorption zu bestehen. Details am linken Bildrand sind schwerer zu erkennen als im unbearbeiteten Bild. Ebenso ist der Zwischenraum am rechten Rand zwischen Kugel und Φ nicht erkennbar. Der rechte Rand der Kugel ist verbessert. Datensatz II filtert auf ähnliche Weise. Das Bildrauschen ist etwas höher als bei Datensatz I. Hierdurch sind die beiden Kugeln besser voneinander trennbar.
Datensatz III reduziert die Streustrahlung lediglich in ihrer Intensität und nicht in ihrem Auftreten. Hierdurch werden die Objektkonturen nicht verbessert und bleiben ungenau. Durch Erhaltung des Bildrauschens bleiben die Materialunterschiede erkennbar. Gleiches lässt sich über Datensatz IV sagen.



(a) Original **(b)** Datensatz I **(c)** Datensatz II

(d) Ground Truth **(e)** Datensatz III **(f)** Datensatz IV

Abbildung 8.20 Es wird ein Ausschnitt des Validierungsphantoms d im Original in Abbildung (a), nach den Datensätzen I, II, III und IV gefiltert in (b), (c), (e) und (f) und als Ground Truth in Abbildung (d) dargestellt.

8.4 Unterschiede der Filterung aufgrund der verschiedenen Datensätze

Die vorangegangenen Abschnitte zeigen die Ergebnisse der Filterung des neuronalen Netzes. Die Güte der Filterung variiert sowohl mit den Iterationen zur Optimierung der Gewichte (vgl. 8.1.2), als auch mit den verschiedenen Trainingsdatensätzen.
Die Filterung wurde sowohl durch Anwendung auf die Trainingsdaten selbst, durch Anwendung auf Testdaten, welche den Trainingsdaten sehr ähnlich sind, und durch Anwendung auf Validierungsdaten analysiert. Aufgrund der hohen Ähnlichkeit zwischen Test- und Trainingsdaten ergaben sich nahezu identische Ergebnisse. Auch die Ergebnisse bei Filterung der Validierungsdaten sind für jeden Datensatz mit denen der Trainingsdaten vergleichbar.
Datensatz I reduziert das Bildrauschen in sehr hohem Maße (deutlich stärker als in der Simulation) und verschlechtert gleichzeitig die Auflösung feiner Strukturen, hauptsächlich in der horizontalen Achse. Datensatz II erhält das Bildrauschen auf dem Niveau, in welchem es auch in der Simulation enthalten ist. Feine Strukturen werden ähnlich zum Training mit Datensatz I verschlechtert dargestellt. Die Validierungsdaten zeigen, dass Datensatz II die Strukturen in geringerem Umfang verbreitert (vgl. Abbildung 8.19) als Datensatz I. Die Korrektur mehrere Pixel umfassender Artefakte gelingt beiden Datensätzen nicht vollständig. Durch das höhere Bildrauschen erreicht das Ergebnis von Datensatz II nahezu die Qualität der Ground Truth (vgl. Abbildung 8.16).
Das bei Datensatz II erhaltene Rauschen enstrpicht in der Größenordnung dem statistischen Rauschen der Simulation. Die Filterung durch Datensatz II liefert somit bessere Ergebnisse als Datensatz I.
Die Datensätze III und IV erzeugen ebenfalls Projektionen mit geringerem Rauschen als Ground Truth, dass heißt auch hier wird das statistische Rauschen reduziert. Im Gegensatz zu den Da-

tensätzen I und II werden hier auch mehrere Pixel umfassende Artefakte reduziert oder entfernt. Strukturen in der Pixelgröße bleiben erhalten, werden allerdings nicht merklich verbessert. Die Streustrahlung wird nicht gänzlich entfernt.

Es ist festzuhalten, dass die Datensätze III und IV vergleichbare Ergebnisse erzeugen. Eine explizite Analyse der Gewichte der zwei Netze könnte hier Gemeinsamkeiten und Unterschiede der trainierten Netze aufzeigen. Es ist sowohl möglich, dass ähnliche Netze erzeugt wurden, als auch, dass zwei unterschiedliche Systeme erzeugt wurden, die dennoch vergleichbare Ergebnisse produzieren. Eine weiterführende Analyse wird nicht durchgeführt.
Die Datensätze I und II erzeugen ebenfalls ähnliche Ergebnisse, allerdings sind hier vor allem beim Bildrauschen große Unterschiede vorhanden. Die Datensätze I und II unterscheiden sich durch die Projektionszahl, die Statistik (Anzahl simulierter Photonen) und den Objektorientierungen bei der Durchleuchtung.
Die Ergebnisse der Datensätze I und II unterscheiden sich von den Ergebnissen der Datensätze III und IV. Die Datensätze I und II bilden Phantome aus verschiedenen Materialien ab, während die Datensätze III und IV lediglich Aluminium umfassen. Nach Abschnitt 2.2.1 ist bekannt, dass die Streuung materialabhängig ist. Insofern ist eine unterschiedliche Streuverteilung zu erwarten, welche das Netz anders trainiert. Es liegt nahe, dass das Material in den Trainingsdaten einen deutlich größeren Einfluss auf die (Filterung der) Streustrahlung hat, als die Geometrie der Objekte. Die Datensätze III und IV umfassen geometrisch sehr unterschiedliche Phantome, erzeugen aber sehr ähnliche Ergebnisse.

Die Reduktion des Bildrauschens ist eine der Aufgaben des erstellten neuronalen Netzes. Allerdings gilt es hier lediglich Rauschen, welches aus Streustrahlung resultiert, zu entfernen. Statistisches Rauschen zu entfernen ist nicht zwangsläufig schlecht. Da die zugrundeliegenden Regeln nicht bekannt sind, sollte aber das Ergebnis zuerst bewertet und eingeordnet werden. Durch Vergleich mit einem Gauss- oder Median- bzw. einem Tiefpass-Filter ließe sich die Rauschreduktion, die über die Streustrahlung hinausgeht analysieren. Aktuell kann allerdings nicht festgestellt werden, ob das Rauschen zu einem akzeptablen Wert korrigiert wird.
Der Streuanteil eines Pixelwertes soll entfernt werden um die Auflösung zu verbessern. Beim statistischen Rauschen findet eine Abweichung sowohl zu niedrigeren, als auch zu höheren Werten statt. Im Gegensatz zu den umgebenden Pixeln muss hier gemittelt und nicht an niedrigere Werte angeglichen werden. Da in den Pooling-Merkmalen, welche dem Netz übergeben werden, sowohl Mittelwerte, als auch Minimal- und Maximalwerte enthalten sind, kann das Netz nach all diesen Kriterien entscheiden und anpassen.
Das das statistische Rauschen reduziert wird zeigt allerdings auch die mögliche Verallgemeinerung, die ein solches Netz leisten kann. Da die Streustrahlung eine Form des Rauschens ist wurde dieses Prinzip aus den Trainingsdaten in Datensatz I verallgemeinert, was zu einer Reduktion des Bildrauschens, unabhängig von seiner Entstehung führt. Solche Verallgemeinerungen sind häufig gewünscht, wenn neuronale Netze trainiert werden.

8.5 Bewertung des neuronalen Netzes

Eine abschließende Bewertung des neuronalen Netzes kann aufgrund des geringen Umfangs und der geringen Tiefe der Untersuchungen nicht angestellt werden. Vor dem Hintergrund des simplen, nicht aufwändig qualitativ erörterten Aufbaus des neuronalen Netzes, kann allerdings eine Abschätzung und Einordnung der Möglichkeiten stattfinden.
Bereits nach einem Training über zwei Epochen ist eine Reduktion der Streustrahlung durch das neuronale Netz möglich (zu berücksichtigen ist hier die in Abschnitt 7.3.4 definierte `batchsize`). 100 Iterationen zeigen eine weitere deutliche Verbesserung (vgl. Abbildung 8.3).
Durch verschiedene Projektionen als Trainingsdaten ergeben sich unterschiedliche Netze. Das durchleuchtete Material in den Trainingsdaten scheint hier einen größeren Einfluss zu haben als die Geometrie des Objektes (vgl. Abschnitt 8.4). Durch die unterschiedlichen Trainingsdatensätze ergeben sich verschiedene Schwächen und Stärken des Netzes. Hier sind die Reduktion des statistischen Rauschens und die Detailsauflösung zu nennen.
Durch eine aufwändigere Analyse durch Variation einzelner Parameter in den Trainingsdaten oder durch Kombination der Datensätze ließen sich möglicherweise reproduzierbare Zusammenhänge finden. Auf Basis solcher Zusammenhänge kann dann ein Netz für eine spezielle Anforderung erstellt werden.
Das trainierte neuronale Netz bietet die Möglichkeit, ein Bild schnell zu filtern. Ob hierdurch ein Vorteil gegenüber herkömmlichen (analytischen) Filtern erreicht wird, ist nicht untersucht. Ein solcher Vergleich zur weiteren Bewertung des Netzes ist sinnvoll. Besteht die Möglichkeit mit analytischen Methoden gleichwertige Ergebnisse zu erhalten, ist dies vorzuziehen, da ein neuronales Netz auch kausal nicht begründbare Zusammenhänge enthalten kann (siehe hierzu Abschnitt 9).
Eine Optimierung der Hyperparameter ist essentiell, um ein neuronales Netz zu optimaler Leistung zu bringen. Eine solche Optimierung hat hier nicht stattgefunden.
Als weiterer Aspekt zur Bewertung sind die Simulationsdaten zu betrachten. Die Zieldaten sind durch Abschätzung mit eine Schranke ermittelt (siehe Abschnitt 6.4.3). Hier wäre eine genauere Bestimmung der Zieldaten möglich. Das heißt, dass die Zielprojektionen, die hier als Ground Truth verwendet werden, nicht unbedingt sämtliche Streustrahlung umfassen. Insofern fehlen dem Netz zum einen Information für ein optimales Training und zum anderen kann nicht untersucht werden, inwiefern zusätzlich durch das Netz erkannte Streustrahlung (im Vergleich zu Ground Truth) eine Verschlechterung oder eine gewünschte Verallgemeinerung darstellt.
Die Simulationen erzeugen Röntgenprojektionen mit sehr geringer Statistik im Vergleich zu den in der Industrie und Medizin üblichen Aufnahmen. Diese geringe Statistik erschwert das Erlernen von Strukturen und ermöglicht keine Anwendung auf reale Projektionen.
Ein Training mit realistischeren Projektionen kann neue Aspekte aufzeigen und andere Ergebnisse liefern. Für Projektionen anderer Statistik müssten die Möglichkeiten des Netzes erneut analysiert werden.

9 Analyse der Anwendbarkeit, der Einschränkungen und der Optimierungsmöglichkeiten

Die Erstellung eines neuronalen Netzes zur Unterstützung in der Datenauswertung, zur Verbesserung der Messgenauigkeit oder zur Einordnung von Ergebnissen ist eine vielversprechende Methode. Im Folgenden wird erörtert, unter welchen Bedingungen diese Methode angewendetet werden kann und sollte. Vor diesem Hintergrund wird das erstellte Netz eingeordnet.

9.1 Anforderungen an ein neuronales Netz zur Anwendung auf ein physikalisches Problem

Bei der Anwendung und Erstellung eines neuronalen Netzes muss berücksichtigt werden, dass neuronale Netze sämtliche Entscheidungen, Kategorisierungen und Vorhersagen nur aus statistischen Gründen treffen können. Möchte man physikalische Gesetzmäßigkeiten durch ein neuronales Netz erfassen, ist es sinnvoll, kausale Zusammenhänge von rein statistischen zu unterscheiden. Werden in Trainingsdaten statistische Zusammenhänge erkannt und das Netz dementsprechend angepasst, werden vor diesem Hintergrund die Klassifizierungen vorgenommen. Bei einem methodischen Fehler in einer Datensimulation kann es zum Beispiel zu statistischen Zusammenhängen kommen, die keinerlei physikalische Bedeutung haben. Dies kann bei der Anwendung des neuronalen Netzes auf reale Daten zu Fehlklassifizierungen führen. Solche statistischen Zusammenhänge können auch in den Validierungsdaten und realen Daten vorkommen, sodass der Mangel an Allgemeinheit zunächst nicht erkennbar ist. Man muss zunächst davon ausgehen, dass es immer zu Fehlklassifizierungen kommen kann, die keine kausalen, sondern rein statistische Gründe haben. Im Folgenden sollen Grundzüge einer Strategie erörtert werden, wie die Sinnhaftigkeit und allgemeine Anwendbarkeit eines neuronalen Netzes bewertet werden kann. Einschränkungen in der Anwendbarkeit können unter anderem auf die zugrundeliegenden Trainingsdaten, auf die Netzarchitektur und die Wahl der Merkmale zurückzuführen sein. In der Anwendung eines neuronalen Netzes werden Aussagen über ein System oder Objekt getroffen. Ein Verständnis für die Genauigkeit und Sicherheit dieser Aussagen ist essentiell wichtig, um auf Basis dieser Aussagen weitere Schlussfolgerungen oder Handlungen zu begründen.

9.1.1 Wahl und Erstellung der Trainingsdaten

Trainingsdaten für neuronale Netze können simuliert oder gemessen werden. Beide Arten der Datenerzeugung beinhalten systematische und statistische Fehler. Es ist also notwendig, die einfließenden Fehlerquellen bzgl. der Ursache (z. B. physikalische Eigenschaft) und der Auswirkung (z. B. ungewollt unterschiedlich starker Einfluss einzelner Paramater) zu analysieren.
Werden die Daten gemessen, findet meist eine aufwändige Analyse der systematischen Fehler statt. Hier wird versucht, den Einfluss jeder Komponente eines Versuchsaufbaus zu erfassen. Im Optimalfall sind alle systematischen Fehlerquellen bekannt und können bei der Datenauswertung berücksichtigt werden. Gerade bei aufwändigen Messungen ist es allerdings schwierig, viele

Messdaten unter identischen Bedingungen zu erzeugen. Infolgedessen sind meist nur wenige Messungen möglich und die statistische Ungenauigkeit steigt.
Die Simulation von Messdaten bietet hauptsächlich den Vorteil, in verhältnismäßig kurzer Zeit viele Daten unter gleichbleibenden Bedingungen erzeugen zu können. Systematische Fehlerquellen im klassischen Sinne werden ausgeschlossen und durch Modellannahmen ersetzt. Die Kosten und der Zeitaufwand der Erstellung und Verwendung eines Versuchsaufbaus werden durch Rechenzeit und die Erstellung einer Simulation ersetzt. Durch die hohe Anzahl durchführbarer Simulationen sinkt zunächst die statistische Ungenauigkeit. Die systematischen Fehler werden allerdings nicht gänzlich ausgeschlossen, sondern durch Ungenauigkeiten und Fehler in den Modellannahmen ersetzt.
Um physikalische Modelle effizient zu gestalten, werden sie meist für einen speziellen Anwendungsfall optimiert. So werden die für diese Anwendung benötigten Parameter mit hoher Genauigkeit bestimmt, während in anderen Parametern höhere Ungenauigkeiten hingenommen werden. Klassischerweise verwendet man im Machine-Learning sämtliche verfügbaren Daten, um den Algorithmus möglichst viele Zusammenhänge finden zu lassen. Problematisch ist, dass in der Regel die verschiedenen Parameter mit unterschiedlichen Genauigkeiten simuliert werden, sie von dem Algorithmus aber nicht differenziert betrachtet werden.
Es gilt also stets, die systematischen Ungenauigkeiten in allen Daten, die einem lernfähigen Algorithmus zur Verfügung gestellt werden, zu berücksichtigen. Es ist zu untersuchen, inwiefern vermeintlich physikalisch irrelevante Daten Einfluss auf die Klassifizierung ausüben. Es ist weiter zu erörtern, ob diese Daten weggelassen werden sollten, um diese Möglichkeit auszuschließen oder ob eventuell ein vorher nicht bedachter kausaler Zusammenhang besteht.

9.1.2 Wahl der Merkmale

Durch die Anpassung der Gewichte wählt ein neuronales Netz im Verlauf des Trainings selbst aus, welche Merkmale in welchem Maße in die Klassifizierung einfließen. Hier können auch kausal unbegründete Zusammenhänge verstärkt werden. Ist also im Vorhinein bekannt, dass ein Merkmal keinen Einfluss auf die Klassifizierung hat, so sollte man es eventuell nicht mit einfließen lassen.
Ein solches Merkmal kann beispielsweise der Zeitpunkt sein, zu dem eine Messung aufgenommen wurde. Meist wird ein Zeitpunkt dokumentiert und häufig besteht auch ein statistischer Zusammenhang zwischen Zeitpunkt der Messung und dem Ergebnis einer Messung. Dieser ist aber nicht unbedingt durch das Merkmal der Zeit begründet, sondern beispielsweise durch eine zeitliche Änderung der Temperatur. Hier kann das Merkmal Zeit wichtige Informationen liefern, die aber sehr differenziert betrachtet werden müssen. Ist eine solche Betrachtung nicht möglich, sollte in Erwägung gezogen werden, das Merkmal aus dem Lernprozess auszuschließen.
Häufig ist genaues Wissen über die Zusammenhänge bei Erstellung des Netzes nicht vorhanden bzw. das Netz wird mit dem Ziel verwendet, herauszufinden, welche Messgrößen einen Einfluss auf die Klassifizierung haben. Um in diesem Fall zu analysieren, welche Merkmale für die Klassifizierung relevant sind, kann der Algorithmus explizit mit und ohne bestimmte Merkmale trainiert werden.

9.1.3 Wahl der Netzarchitektur

Ein lernender Algorithmus sollte die statistischen und/oder die kausalen (physikalischen) Zusammenhänge in einer Datenmenge erfassen. Neuronale Netze werden meist lediglich anhand statistischer Zusammenhänge trainiert. Statistische Zusammenhänge repräsentieren häufig auch die physikalischen Hintergründe, allerdings ist es nicht immer möglich aus den statistischen Zusammenhängen eine allgemeingültige Regel zu erstellen. Es stellt sich die Frage, welche Allgemeinheit sich durch das Netz erreichen lässt.
Bei der Anwendung auf physikalische Probleme kann die Netzarchitektur auch die Möglichkeit bieten, sämtliche Daten auf eine Art und Weise miteinander zu verknüpfen, welche die auftretenden physikalischen Gesetze berücksichtigt. Hierfür sollten die zur Beschreibung der Physik notwendigen mathematische Methoden eingebaut und die Merkmale entsprechend gewählt sein. Die Netzarchitektur setzt sich aus der Anzahl der Schichten sowie den Verknüpfungen der Schichten zusammen. Es lassen sich die physikalisch begründeten mathematischen Operationen auf einzelnen oder mehreren Merkmalen ausführen, um physikalische Zusamenhänge erkennbar zu machen.

9.1.4 Wahl der Validierungsdaten

Die Funktionalität eines trainierten Algorithmus zeigt sich bei der Anwendung auf Validierungsdaten. Die Validierungsdaten sollten somit jeden Anwendungsbereich abdecken. Dadurch kann nach einer Bewertung möglicherweise eine Eignung für gewisse Bereiche gegeben sein, für andere Bereiche jedoch nicht. Es ist nicht unbedingt trivial, Validierungsdaten zu erzeugen, die alle möglichen Eigenschaften von zukünftig zu klassifizierenden Daten aufweisen. Insofern ist es besonders wichtig, die individuellen Eigenschaften (beispielsweise systematische Fehlerquellen, zugrundeliegende Annahmen), die die Trainingsdaten und die Validierungsdaten aufweisen zu dokumentieren und zu diskutieren.

9.2 Einordnung des erstellten Netzes

Das in dieser Arbeit vorgestellte neuronale Netz wird mit simulierten Daten trainiert. Die Simulation basiert auf Monte Carlo-Methoden und kann als sehr exakt angenommen werden. Die größte Ungenauigkeit, welche in die Trainingsdaten einfließt, entsteht durch die Art der Filterung zur Erzeugung der Ground Truth-Daten. Die Filterung findet nicht auf Basis der physikalischen Effekte, sondern aufgrund geometrischer Überlegungen statt. Des Weiteren wird lediglich eine Abschätzung durch eine Schranke vorgenommen (siehe Abschnitt 6.4.3). Der Einfluss dieser Abschätzung wird nicht analysiert. Stattdessen wird diese Filterung als Ground Truth und somit als Referenz für die Güte der Filterung gewählt.
Das Netz wurde nicht darauf ausgerichtet, die physikalischen Zusammenhänge zu modellieren. Die dem Netz übergebenen Merkmale werden nicht explizit ausgewählt um die Physik zu repräsentieren. Die Pooling-Merkmale repräsentieren lediglich statistische Eigenschaften und haben keine direkte physikalische Bedeutung, lediglich eine physikalische Ursache. Da die Entstehung von Streustrahlung als eine Faltungsoperation beschrieben werden kann, ist den Faltungs-Merkmalen eine physikalische Bedeutung zuzuschreiben (siehe hierzu die Abschnitte 7.2.2 und 9.2.1). Da die verwendeten Streukerne anhand ihrer optischen Funktion ausgewählt wurden und nicht aufgrund

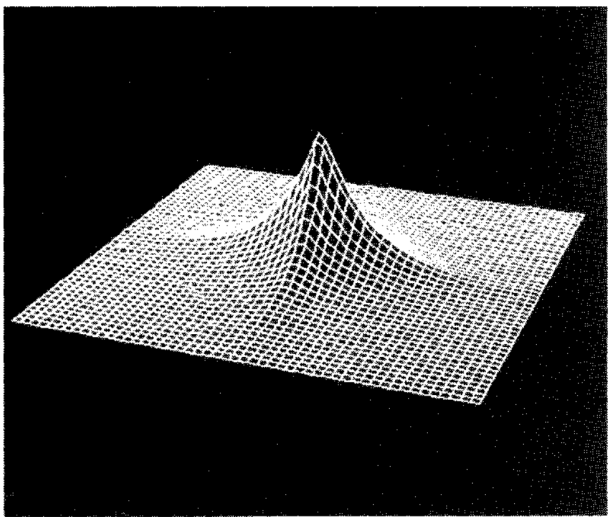


Abbildung 9.1 Darstellung einer zweidimensionalen Punktspreizfunktion zur Bestimmung des Rauschanteils.[32, S. 182]

ihrer Erfassung von Streustrahlung, sollten auch diese Merkmale hauptsächlich als statistisch begründet gewertet werden.
Vor diesem Hintergrund wäre eine Analyse der Bedeutung der einzelnen Merkmale für die Klassifizierung interessant. Eine solche Analyse kann zunächst durch Weglassen einzelner Parameter geschehen und durch eine Analyse der trainierten Gewichte erweitert werden. Aufgrund des eingeschränkten Umfangs der Bachelorarbeit findet keine Anaylse dieser Art statt.
Die weitere Architektur sowie die Hyperparameter des neuronalen Netzes wurden nicht erörtert. Die Auswahl des Parameters, als auch der Pooling- und Faltungs-Merkmale, lässt sich als *educated guess* bezeichnen, welcher auf Basis eigener Erfahrungen der Bildfilterung, der gelesenen Literatur und dem Nachvollziehen einiger Beispiele sowie dem Ausprobieren einiger Werte getätigt wurde.
Der einzige Aspekt, dessen Einfluss auf die Ergebenisse des Netzes analysiert wurde, sind die Trainingsdaten selbst. In Abschnitt 8 wird die Streustrahlerkennung des Netzes für vier verschiedene Trainingsdatensätze untersucht. Es fällt auf, dass sowohl die Anzahl der verfügbaren Trainingsdaten einen Einfluss hat, als auch das Material des Phantoms im Training. Letzteres hatte einen deutlich größeren Einfluss auf das Ergebnis des neuronalen Netzes, als die Geometrie des Phantoms. Physikalisch wird eine starkte Abhängigkeit vom Material des Phantoms erwartet, insofern scheint dieser Aspekt durch das Netz repräsentiert zu sein.
Jegliche Bewertung des Netzes ist aufgrund der geringen Anzahl untersuchter Phantome und Projektionen unter Vorbehalt zu betrachten. Um konkrete Aussagen über die Repräsentation der Physik durch das neuronale Netz zu treffen, bedarf es weiterer Untersuchungen.

9.2.1 Mögliche Strategien zur Anpassung des neuronalen Netzes an die zugrundeliegenden physikalischen Gesetzmäßigkeiten

Um in dem Anwendungsfall der Erkennung von Streustrahlung auf physikalischer Basis ein neuronales Netz zu erstellen bzw. das erstellte Netz zu erweitern, müssen die zugrundeliegenden physikalischen Prozesse weiter erörtert werden. Die Entstehung von Streustrahlung wurde in Abschnitt 2.2 kurz dargestellt. Das durchstrahlte Objekt kann explizit als Streukörper betrachtet

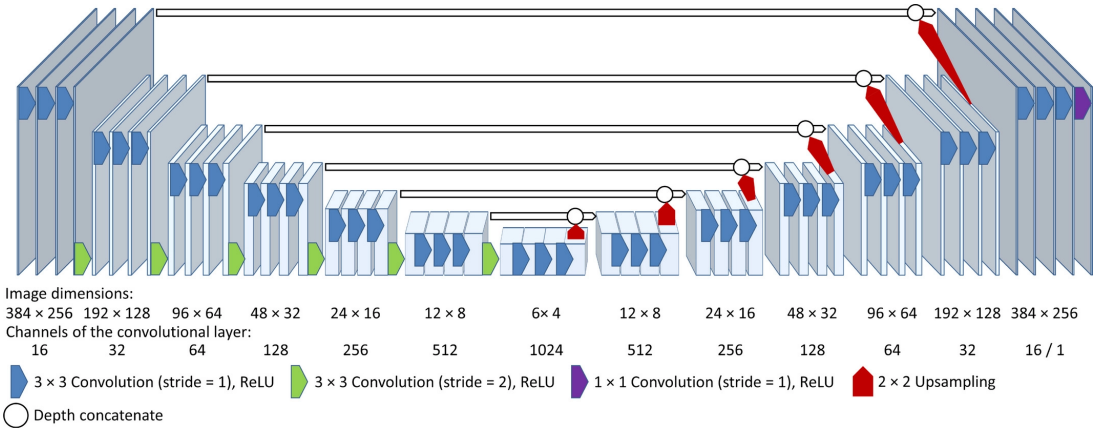


Abbildung 9.2 Aufbau des durch Maier et al. zur medizinischen *Deep scatter estimation* verwendeten neuronalen Netzes [33, S. 239].

werden. Infolgedessen ergeben sich geometrische und materialbezogene Bedingungen. So lassen sich einzelnen Objektpunkten Wirkungsquerschnitte anhand der Materialien und Röntgenenergie zuordnen.
Auf dieser Basis lassen sich Streukern-Ansätze verfolgen [32]. Bei einem solchen Ansatz wird das Bild mit einem zuvor zu bestimmenden Streukern gefaltet. Ein Beispiel eines solchen Kerns ist in Abbildung 9.1 zu sehen. Dieser Kern lässt sich als die Verteilung der gestreuten Photonen auf dem Detektor interpretieren, wenn im Zentrum das ungestreute Photon ist. Eine solche Matrix wird auch als Punkspreizfunktion bezeichnet, da sich ein punktförmiges Signal entsprechend verbreitert.

9.2.2 Vergleich mit aktuellen Entwicklungen

Maier et al. hat 2018 ein Deep Lerning Verfahren zur Streustrahlerkennung sowohl für den medizinischen Bereich als auch den industriellen Bereich [33, 4] vorgestellt. Das Verfahren wird als *Deep scatter estimation* (DSE) bezeichnet und umfasst ein neuronales Netz mit vielen Schichten, welche durch Faltungsoperationen verbunden sind (vgl. Abbildung 9.2). Ziel ist es nicht, einen möglichst guten Streukern auf Basis eines bestimmten Modells zu finden, sondern, dass das Modell selbst durch das Netz definiert bzw. angepasst wird [33, S. 247]. Da das Netz aus diversen 3×3 Faltungsoperationen besteht, verwendet es die grundlegende Methode der Streukerne, ist allerdings deutlich anpassbarer. Dies zeigt sich in den Ergebnissen. Das Netz liefert, die klassischen Streukern-basierten Methoden deutlich übertreffende, Ergebnisse. Dies gilt sowohl, wenn das Netz explizit mit Aufnahmen vergleichbarer Objekte (Röntgenaufnahme des gleichen Körperteils) trainiert wurde, als auch in dem Fall, dass zusätzlich zu den vergleichbaren Objekten unähnliche Aufnahmen (anderer Körperteile) im Training verwendet werden. Das neuronale Netz kann also gleichzeitig für verschiedene Objekte trainiert werden. Die gleiche Aussage lässt sich für Aufnahmen mit verschiedenen Energiespektren treffen. Hierbei sind die Abweichungen zwischen mit gleichem Energiespektrum und mit mehreren Energiespektren trainiertem Netz jedoch größer [33, S. 245].

In den Experimenten von Maier et al. wird weiter verglichen, welchen Einfluss das Training mit verschiedenen Mappings (der Grauwerte) der Projektionsdaten hat. Verglichen werden die reinen Grauwerte p, mit den Mappings e^{-p} und $p \cdot e^{-p}$ als Eingabedaten. Es zeigen sich deutliche

Unterschiede durch das Mapping. Die logarithmisch gemappten Daten trainieren das Netz zu deutlich besseren Ausgaben und einer besseren Verallgemeinbarkeit [33, S. 245, 247].
In den Entwicklungen von Maier et al. basiert das neuronale Netz auf der Bildfaltung. Dadurch, dass über die verschiedenen Ebenen des Netzes diverse Faltungsoperationen angewendet werden, wird kein konkretes Streukern-Modell vorgegeben. Die Anwendbarkeit und vor allem die Allgemeingültigkeit bezüglich der Trainingsdaten wird untersucht. Hier wird sowohl Fokus auf die mögliche Vielfalt der Eingabedaten und Anwendung gelegt, als auch das Mapping der Eingabedaten. Die Struktur des neuronalen Netzes wird nicht variiert. Die Ergebnisse werden mit anderen aktuellen Methoden zur Streustrahlerkennung verglichen. Als Ground Truth werden hier ebenfalls Monte Carlo Simulationen verwendet. Maier et al. verwendet stets herunter-skalierte Bilder. Diese lassen sich schneller verarbeiten und beschränken aufgrund der niedrigen Frequenzen der Streustrahlung nicht dessen Erkennung [4, S. 57].

Rückschlüsse für das entwickelte Netz

Der Ansatz des entwickelten Netzes unterscheidet sich grundlegend von dem Ansatz, welcher durch Maier et al. vorgestellt wurde. Im Vergleich zu Maier et al. findet keine mehrschichtige Faltung statt. Das vorgestellte neuronale Netz verarbeitet lediglich verschiedene, einfach gefaltete oder zusammengefasste (Pooling), den Pixeln zugeordnete Merkmale. Die Verwendung des Grundprinzips der Faltung bleibt eine Gemeinsamkeit.
Das vorgestellte Netz könnte allerdings durch die von Maier et al. erörterten und verwendeten Methoden, unabhängig von der unterschiedlichen Architektur, erweitert werden. Hier ist beispielsweise das Mapping der Grauwerte zu nennen. In Abschnitt 7.2.2 wird beschrieben, dass auf die Daten ein lineares Mapping auf den Wertebereich [0,1] angewendet wird. Maier et al. zeigt, dass verschiedene logarithmische Mappings deutlich bessere Resultate liefern als die lineare Verteilung der Grauwerte [33, S. 245, 247].
Weiter zeigt Maier et al., dass sein Netz durch Training mit verschiedenen Datensätzen auf sämtliche diesen Datensätzen zuordenbare Projektionen mit hoher Güte anwendbar ist. Ein Training mit allen Datensätzen könnte mit dem entwicklten Netz durchgeführt und die Ergebnisse verglichen werden.
Maier et al. arbeitet mit herunter skalierten Daten. Dies ist ein Aspekt, der in dem in dieser Arbeit vorgestellten Netz nicht beachtet wird. Der Einfluss der Skalierung könnte auch an dem präsentierten Netz untersucht werden und potentiell zu Verbesserungen führen. Die Anwendung des Netzes hat, je nach zugrundeliegendem Datensatz, zu einer starken Reduktion des hochfrequenten, statistischen Rauschens geführt. Dieser Effekt kann durch eine andere räumliche Skalierung ausgeschlossen werden.

9.3 Einordnung der verwendeten Methoden

Das vorgestellte neuronale Netz wurde mit PYTHON unter Verwendung grundlegender mathematischer Bibliotheken erstellt (vgl. Abschnitt 6.2). Es existieren diverse Bibliotheken, die explizit auf die effiziente und anwendungsbezogene Implentierung von neuronalen Netzen und Machine-Learning Algorithmen allgemein spezialisiert sind. Zu nennen sind unter anderem TENSORFLOW und SCIKIT-LEARN [34, 35]. SCIKIT-LEARN umfasst viele verschiedene Klassifizierer und

Clustering-Algorithmen. Sogenannte *pipelines* werden verwendet, um die verschiedenen Algorithmen auf Datenmengen anzuwenden und miteinander zu kombinieren.
TensorFlow bietet die Möglichkeit, durch die enthaltene Bibliothek keras, direkt Modelle zum Machine-Learning anhand der jeweils verfügbaren Parameter zu definieren und anzuwenden. Es ist eine sehr effiziente Implementierung vorhanden, die vor allem auf Grafikprozessoren schnelle Berechnungen erlaubt.
Das neuronale Netz dieser Arbeit hätte auch durch eine der oben genannten Bibliotheken erstellt werden können. Durch integrierte Algorithmen zur Hyperparameteranalyse und Bewertung der Ergebnisse ist neben einer schnellen Erstellung auch eine einfache Analyse der Leistung des Netzes möglich.
Nachdem einige Skripte unter Verwendung von TensorFlow und scikit-learn nachvollzogen und getestet wurden, wurde für diese Arbeit die Entscheidung getroffen, die grundlegenden Algorithmen selbst zu entwickeln bzw. nachzuvollziehen und zu implementieren. Hauptsächliches Ziel dieser Bachelorarbeit ist es, einen Überblick und einen Einstieg in die Methoden des Machine Learnings und im speziellen der neuronalen Netze zu erlangen. Durch die selbstständige Implementierung der Algorithmen wurde ein tiefergehendes Verständnis der Methoden sowie ihrer Schwachstellen erlangt.

10 Zusammenfassung

In dieser Arbeit wurden durch Simulation Röntgenprojektionen erzeugt, welche anhand der Simulationsinformationen zu rauschreduzierten Projektionen gefiltert wurden. Mit diesen Paaren rauschreduzierter und unbearbeiteter Projektionen wurde ein neuronales Netz trainiert, welches angewendet wurde, um den Rauschanteil in Röntgenprojektionen zu ermitteln.

Es wurden die Grundlagen der Röntgenbildgebung sowie explizit die für die Streuung von Röntgenstrahlung relevanten Effekte dargestellt (siehe Abschnitt 2). Es wurde die Anwendung der Simulationssoftware GATE dokumentiert und zur Erzeugung von Röntgenaufnahmen vier verschiedener Phantome verwendet (siehe die Abschnitte 3 und 5). Die vier Phantome lieferten vier Datensätze von Projektionen unter 30 (Datensatz I) bzw. 120 verschiedenen Perspektiven (Datensätze II, III und IV). Die Phantome bestanden aus zwei unabhängig voneinander rotierenden Körpern. Die Datensätze unterschieden sich durch die verwendeten Materialien und ihre Verteilung im Phantom sowie die Rotation der beiden Körper zueinander. Auf Basis der Simulationsdaten wurde ein Verfahren zur Erzeugung rauschreduzierter Projektionen entwickelt. Zusätzlich wurden acht weitere Phantome erstellt, welche später zur Bewertung des neuronalen Netzes dienten.
Durch eine Filterung anhand der zurückgelegten Weglänge der Röntgenphotonen wurden rauschreduzierte Projektionen erzeugt. Es wurden auch Filterungen durch Energiefenster und durch die in der Simulation registrierten Streuprozesse getestet. Die besten Ergebnisse zeigten sich durch die geometrische Filterung auf Basis der Weglänge (vgl. Abschnitt 6.4).
In Abschnitt 4 wurden die grundlegenden Elemente und Konzepte neuronaler Netze erörtert. Hierbei wurden auch Probleme wie die Über- und Unteranpassung und weiterführende Konzepte wie die Modulation der Hyperparameter und die Merkmalselektion und -extraktion erarbeitet. Abschnitt 7 beschreibt die Implementierung der Methoden für den gewählten Anwendungsfall.
Es wurde ein mehrschichtiges neuronales Netz vorgestellt, welches die Projektionen pixelweise mit pixelweisen Merkmalen verarbeitet. Diese Merkmale wurden durch Pooling- und Faltungs-Operationen aus dem Pixelwert und den Umgebungspixeln bestimmt.
Es wurden zur Datenverarbeitung, zur Merkmalserstellung und als neuronales Netz drei Klassen erstellt. Es wurden mehrere Skripte zur Datenvorverarbeitung und Filterung entwickelt. Dadurch konnten die Simulationen parallelisiert werden und die Simulationsdaten in durch PYTHON verarbeitbare Arrays geschrieben werden (siehe hierzu Abschnitt 6). Die Implementierung sämtlicher Methoden ist mit PYTHON erfolgt.
In Abschnitt 8 wurde die Anwendung des neuronalen Netzes untersucht. Zunächst wurde das Netz auf die zum Trainieren verwendeten Projektionen der vier Datensätze angewendet, um festzustellen, wie viele Iterationen des Trainings notwendig sind, um eine Verbesserung der Projektionen zu erreichen und ob eine Über- oder Unteranpassung stattgefunden hat. Das neuronale Netz wurde bis zu 1000 Iterationen trainiert und zeigte eine - sich mit der Iterationszahl verbessernde - Filterung konnte die Streustrahlung jedoch nie vollständig filtern. Es fand also keine direkte Überanpassung statt und es ist zu vermuten, dass das Netz die nötigen Informationen, um das Rauschen der Traininsgsdaten vollständig zu charakterisieren nicht aufnehmen konnte.

Es wurde untersucht, ob die Filterung unbekannter Validierungsdaten Ergebnisse gleicher Qualität liefert. Das Netz wurde mit den vier Datensätzen trainiert und erreicht sehr unterschiedliche Ergebnisse. Große Teile der Streustrahlung wurden herausgefiltert. Anhand verschiedener Phänomene wurde die Qualität der Filterung optisch eingeordnet. Zum einen wurde hochfrequentes Bildrauschen, was als statistisches Rauschen verstanden werden kann, durch das neuronale Netz aus den Projektionen entfernt. Hierdurch wurden feine Strukturen verschlechtert und dem Bild Objektinformation entzogen. Zum anderen konnten auf Streustrahlung basierte Artefakte und daraus resultierende Fehler in der Objektdarstellung unterschiedlich gut herausgefiltert werden. Beim Vergleich der verschiedenen Datensätze fiel auf, dass die Filterung von dem Material des Phantoms in den Trainingsdaten und von der Anzahl der zum Training verwendeten Projektionen abhängig ist.
Im letzten Abschnitt (9) wurde der Ansatz des erstellten neuronalen Netzes anhand physikalischer Anforderungen bewertet. Es wurde erörtert, inwiefern in den einzelnen Bereichen der Merkmalauswahl, der Netzarchitektur sowie der Wahl der Trainings- und Validierungsdaten die physikalische Problemstellung berücksichtigt werden kann und sollte.
Des Weiteren wurden aktuelle Entwicklungen vorgestellt, die vergleichbare Ansätze zu dem in dieser Arbeit vorgestellten neuronalen Netz verfolgen.

10.1 Ausblick

Es bleibt festzustellen, dass die vorgestellte Methode nur einen Bruchteil der im Bereich der neuronalen Netze und im Bereich des maschinellen Lernens verfügbaren Methoden ausnutzt. Des Weiteren wurden die physikalischen Grundlagen des nachzuvollziehenden Systems nicht adäquat berücksichtigt. Auch an dieser Stelle besteht Potential die vorgestellte Methode zu optimieren. Ebenso stehen zu der Implementierung deutlich weitreichendere, umfangreichere und anpassbarere Methoden in verschiedenen Bibliotheken zur Verfügung [19, 35, 34].
Die Resultate der vorgestellten Methode weisen erhebliche Schwachstellen auf. Auch aufgrund mangelnder, umfassender Analyse der Ergebnisse kann die Methode nicht als anwendbar bezeichnet werden. Es können für keinen Anwendungsbereich grobe Fehler und Ungenauigkeiten ausgeschlossen werden.
Dennoch sind die erreichten Resultate bemerkenswert. Sowohl die Erzeugung der Trainingsdaten, als auch die Auswahl der Hyperparameter, einschließliche der Netzarchitektur, sind nicht systematisch erfolgt. Dennoch wird eine Reduktion der Streustrahlung erreicht. Wichtig ist festzuhalten, dass dabei Nebeneffekte auftreten, die nicht abschließend analysiert wurden.

Literatur

[1] Prof. Dr. H. Bomsdorf. *Bildgebende Verfahren*. Vorlesungsskript. 2017.

[2] Simon R. Cherry, James A. Sorenson und Michael E. Phelps. „chapter 16 - Tomographic Reconstruction in Nuclear Medicine“. In: *Physics in Nuclear Medicine (Fourth Edition)*. Hrsg. von Simon R. Cherry, James A. Sorenson und Michael E. Phelps. Fourth Edition. W.B. Saunders, 2012, S. 253 –277.

[3] Ernst-Peter Rührnschopf and und Klaus Klingenbeck. „A general framework and review of scatter correction methods in cone beam CT. Part 2: Scatter estimation approaches“. In: *Medical Physics* 38.9 (2011), S. 5186–5199.

[4] Joscha Maier u. a. „Deep Scatter Estimation (DSE): Accurate Real-Time Scatter Estimation for X-Ray CT Using a Deep Convolutional Neural Network“. In: *Journal of Nondestructive Evaluation* 37.3 (2018), S. 57. ISSN: 1573-4862.

[5] W. C. Röntgen. „On a New Kind of Rays“. In: *Science* 3.59 (1896), S. 227–231. ISSN: 00368075, 10959203. URL: http://www.jstor.org/stable/1623595.

[6] Simon R. Cherry, James A. Sorenson und Michael E. Phelps. „chapter 2 - Basic Atomic and Nuclear Physics“. In: *Physics in Nuclear Medicine (Fourth Edition)*. Hrsg. von Simon R. Cherry, James A. Sorenson und Michael E. Phelps. Fourth Edition. W.B. Saunders, 2012, S. 7 –18.

[7] Simon R. Cherry, James A. Sorenson und Michael E. Phelps. „chapter 6 - Interaction of Radiation with Matter“. In: *Physics in Nuclear Medicine (Fourth Edition)*. Hrsg. von Simon R. Cherry, James A. Sorenson und Michael E. Phelps. Fourth Edition. W.B. Saunders, 2012, S. 63 –85.

[8] *Website der* INTERNATIONAL OPENGATE COLLABORATION. URL: http://www.opengatecollaboratio
org/home (besucht am 01. 02. 2019).

[9] S Jan u. a. „GATE: a simulation toolkit for PET and SPECT“. In: *Physics in Medicine and Biology* 49.19 (2004), S. 4543–4561.

[10] S Jan u. a. „GATE V6: a major enhancement of the GATE simulation platform enabling modelling of CT and radiotherapy“. In: *Physics in Medicine and Biology* 56.4 (2011), S. 881–901.

[11] *GitHub der* GATE *Software*. URL: https://github.com/OpenGATE/Gate (besucht am 01. 02. 2019).

[12] *Wiki der* GATE *Software*. URL: http://wiki.opengatecollaboration.org/index.php/Main_Page (besucht am 01. 02. 2019).

[13] *Website des* GEANT4 *Toolkits*. URL: http://www.geant4.org/geant4/ (besucht am 01. 02. 2019).

[14] *Website der* ROOT *Software*. URL: https://root.cern.ch/ (besucht am 01. 02. 2019).

[15] *Wiki der* GATE *Software - Abschnitt: Users Guide:Setting up the Physics*. URL: http://wiki.opengatecollaboration.org/index.php/Users_Guide:Setting_up_the_physics (besucht am 01. 02. 2019).

[16] Warren S. McCulloch und Walter Pitts. „A logical calculus of the ideas immanent in nervous activity“. In: *The bulletin of mathematical biophysics* 5.4 (1943), S. 115–133. ISSN: 1522-9602.

[17] F. Rosenblatt. *The Perceptron, a Perceiving and Recognizing Automaton Project Para*. Report: Cornell Aeronautical Laboratory. Cornell Aeronautical Laboratory, 1957.

[18] Steve Parker. *ompaktatlas Menschlicher Körper*. Dorling Kindersley Limited, 2014. ISBN: 978-3-8310-2548-0.

[19] Sebastian Raschka, Vahid Mirjalili und Knut Lorenzen. *Machine learning mit Python und Scikit-learn und TensorFlow: das umfassende Praxis-Handbuch für Data Science, Deep Learning und Predictive Analytics; Python machine learning*. 2., aktualisierte und erweiterte Auflage. Frechen: mitp, 2018. ISBN: 978-3-95845-733-1; 3-95845-733-9.

[20] Bernard Widrow und Marcian E Hoff. *Adaptive switching circuits*. Techn. Ber. Stanford Univ Ca Stanford Electronics Labs, 1960.

[21] *Wiki der* GATE *Software - Abschnitt: Users Guide:Source*. URL: http://wiki.opengatecollaboration.org/index.php/Users_Guide:Source (besucht am 01. 02. 2019).

[22] G. Hernández und F. Fernández. *XPECGEN Software*. URL: https://github.com/Dih5/xpecgen (besucht am 23. 02. 2019).

[23] G. Hernández und F. Fernández. „A model of tungsten anode x-ray spectra“. In: *Medical Physics* 43.8Part1 (2016), S. 4655–4664.

[24] NUMPY *Website*. URL: https://www.numpy.org/ (besucht am 11. 02. 2019).

[25] SCIPY *Website*. URL: https://www.scipy.org/ (besucht am 11. 02. 2019).

[26] *Website des* PYROOT *Pakets*. URL: https://root.cern.ch/pyroot (besucht am 15. 02. 2019).

[27] MATPLOTLIB *Website*. URL: https://matplotlib.org/ (besucht am 06. 03. 2019).

[28] PILLOW *project overview*. URL: https://pypi.org/project/Pillow/ (besucht am 07. 03. 2019).

[29] Nick Cortale. *Standard Deviation (Filters) in Matlab and Python*. URL: https://nickc1.github.io/python,/matlab/2016/05/17/Standard-Deviation-(Filters)-in-Matlab-and-Python.html (besucht am 11. 02. 2019).

[30] Andreas Maier u. a. *Medical Imaging Systems Elektronische Ressource: An Introductory Guide*. Image Processing, Computer Vision, Pattern Recognition, and Graphics, Teil 11111. Springer International Publishing, Imprint: Springer, 2018. ISBN: 978-3-319-96520-8.

[31] Rumelhart David E., Hinton Geoffrey E. und Williams Ronald J. „Learning representations by back-propagating errors“. In: *Nature* 323 (Okt. 1986), S. 533.

[32] L. Alan Love und Robert A. Kruger. „Scatter estimation for a digital radiographic system using convolution filtering“. In: *Medical Physics* 14.2 (1987), S. 178–185.

[33] Joscha Maier u. a. „Real-time scatter estimation for medical CT using the deep scatter estimation: Method and robustness analysis with respect to different anatomies, dose levels, tube voltages, and data truncation“. In: *Medical Physics* 46.1 (2019), S. 238–249.

[34] TensorFlow *Website*. URL: https://www.tensorflow.org/ (besucht am 19. 03. 2019).

[35] scikit-learn *Website*. URL: https://scikit-learn.org/stable/ (besucht am 19. 03. 2019).

A Anhang

A.1 Weitere Projektionen

A.1.1 Trainingsdaten

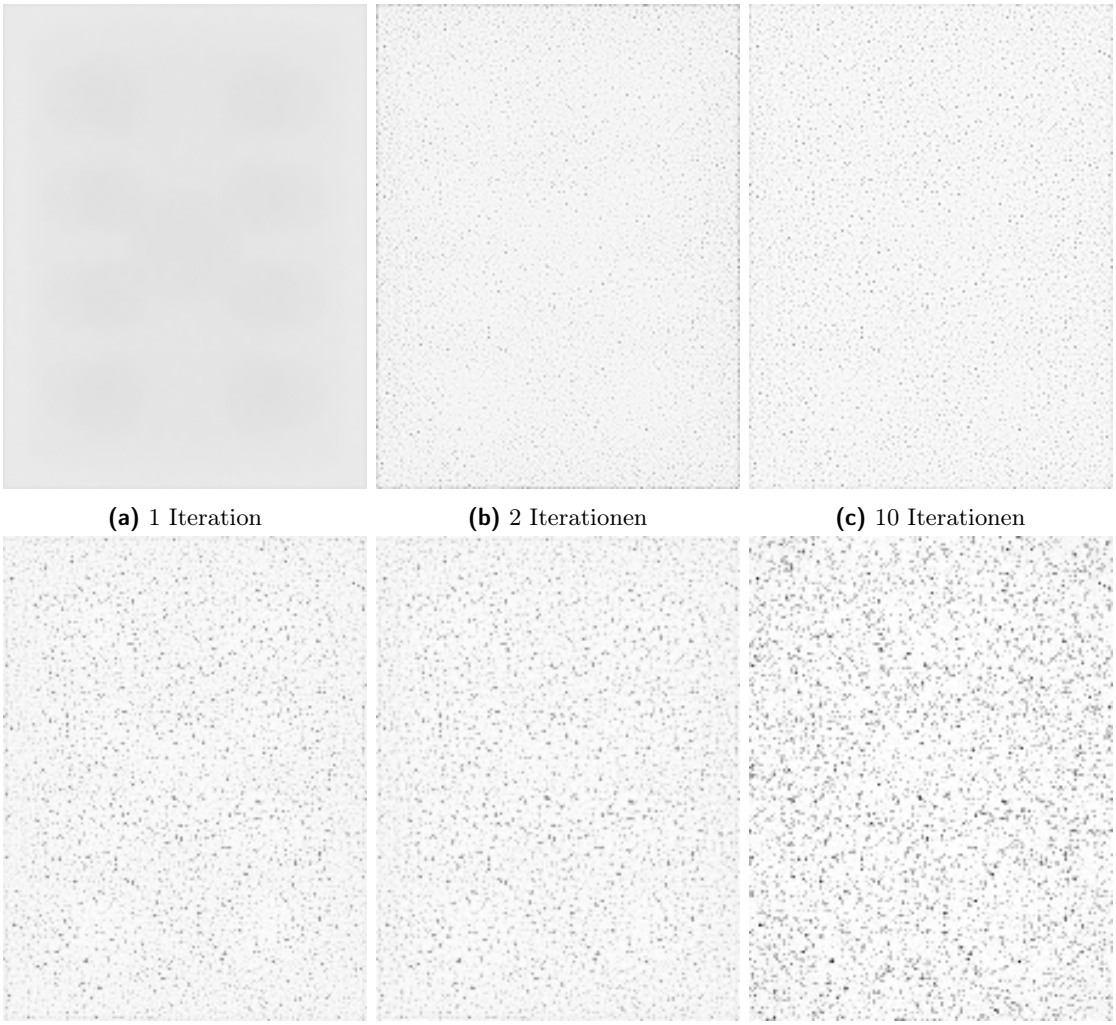
Abbildung A.1 ziegt die unverstärkten Streubilder aus Abbildung 8.4 in Abschnitt 8.1

(a) 1 Iteration **(b)** 2 Iterationen **(c)** 10 Iterationen

(d) 100 Iterationen **(e)** 1000 Iterationen **(f)** Ground Truth

Abbildung A.1 Bild (a), (b), (c), (d) und (e) zeigen den bestimmten Rauschanteil der Projektion (Datensatz III) nach 1, 2, 10, 100 und 1000 Iterationen des neuronalen Netzes. (f) zeigt den, aus den Simulationsdaten vorhergesagten Rauschanteil. Die Darstellung verwendet die invertierten Grauwerte, um einen größeren Kontrast zu bieten.

Abbildung A.2 zeigt die Streumatrizen aus Abbildung 8.4 ohne einen Gamma-Faktor ($\gamma = 1$).



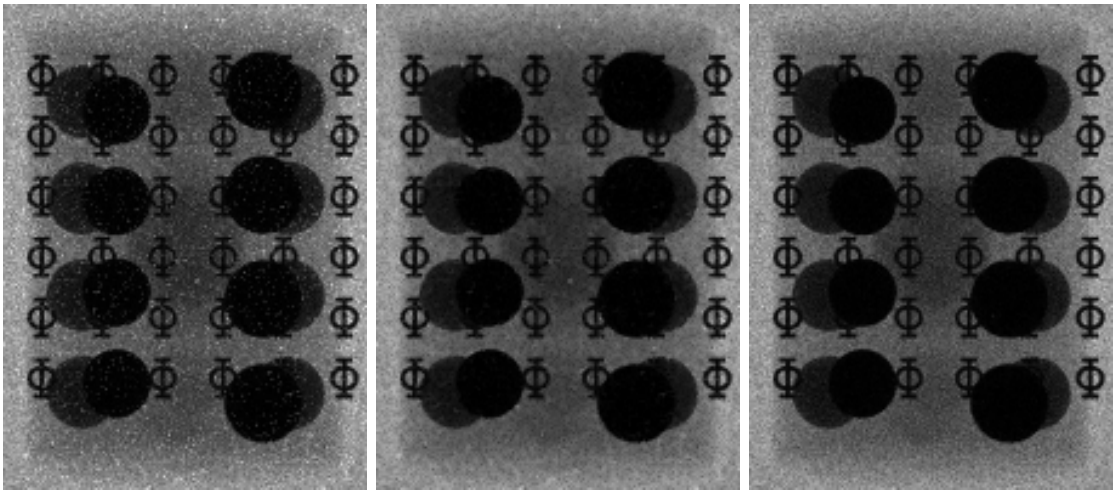
(a) 1 Iteration **(b)** 2 Iterationen **(c)** 10 Iterationen

(d) 100 Iterationen **(e)** 1000 Iterationen **(f)** Ground Truth

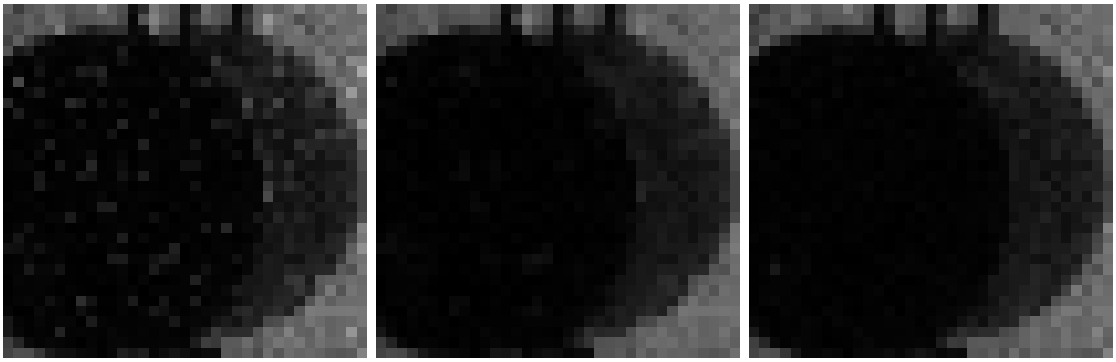
Abbildung A.2 Bild (a), (b), (c), (d) und (e) zeigen den bestimmten Rauschanteil der Projektion (Datensatz III) nach 1, 2, 10, 100 und 1000 Iterationen des neuronalen Netzes. (f) zeigt den, aus den Simulationsdaten vorhergesagten Rauschanteil. Die Abbildung zeigt Rauschanteile relativ zum angestrebten Bild (f) skaliert. Die unskalierte Abbildung ist im Anhang unter A.1 in Abschnitt A.1.1 einzusehen. Die Darstellung verwendet die invertierten Grauwerte, um einen größeren Kontrast zu bieten.

A.1.2 Testdaten

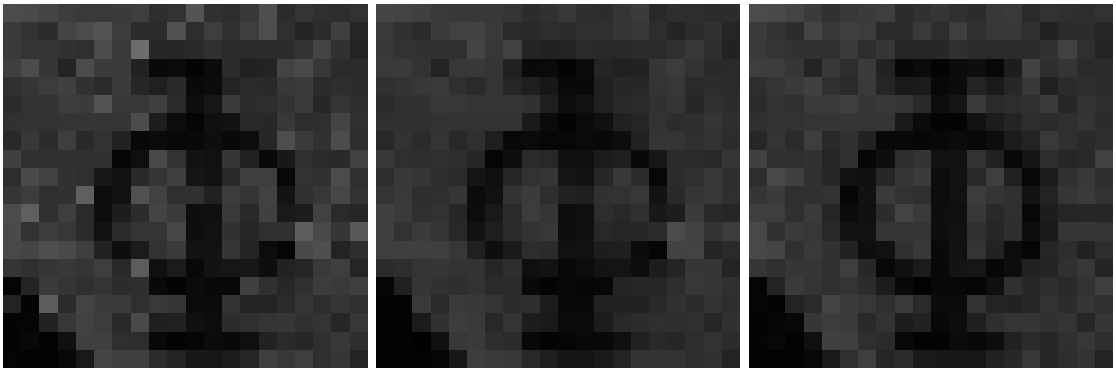
Datensatz I - Testdaten



(a) Original **(b)** 1000 Iterationen **(c)** Ground Truth

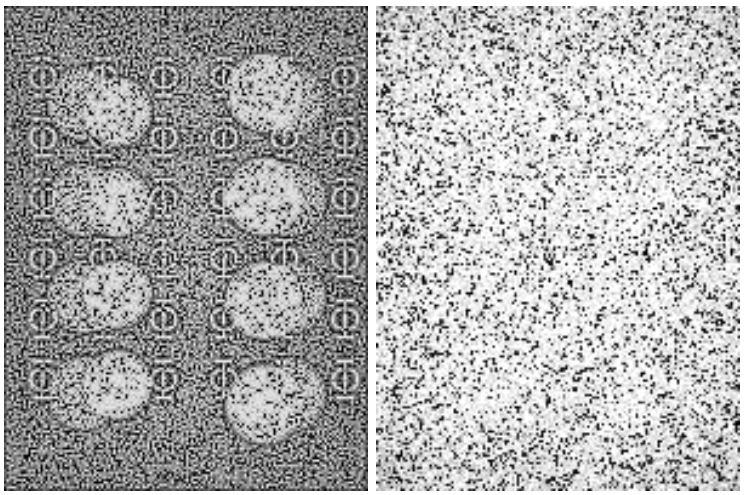


(d) Original **(e)** 1000 Iterationen **(f)** Ground Truth



(g) Original **(h)** 1000 Iterationen **(i)** Ground Truth

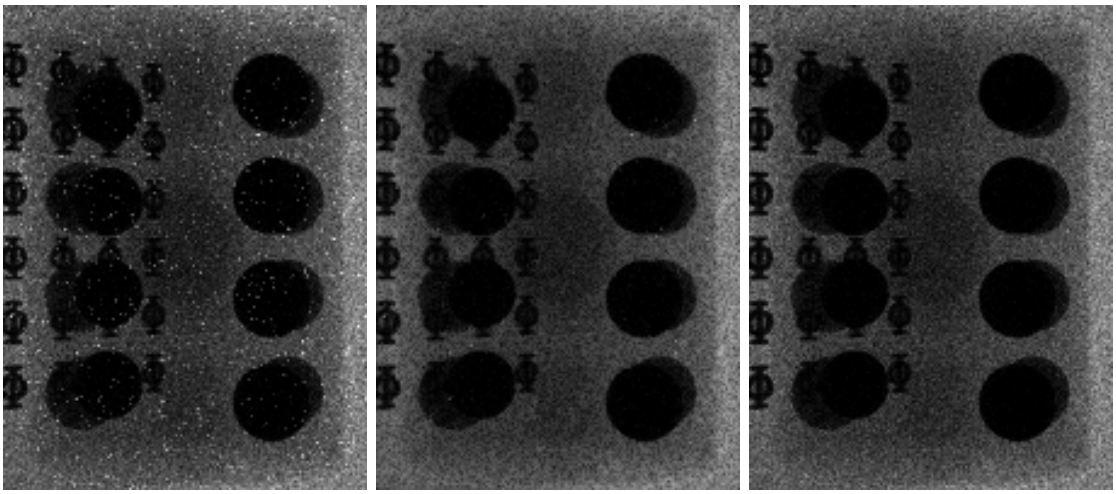
Abbildung A.3 Zu sehen ist eine Projektion aus den Testdaten des Datensatzes I, inklusive zweier Details. Die Abbildungen (a), (d) und (g) sind der unbearbeiteten Projektion entnommen. (b), (e) und (h) wurden durch das über 1000 Iterationen trainierte neuronale Netz gefiltert. (c), (f) und (i) wurden auf Basis der Simulationsdaten rauschreduziert.



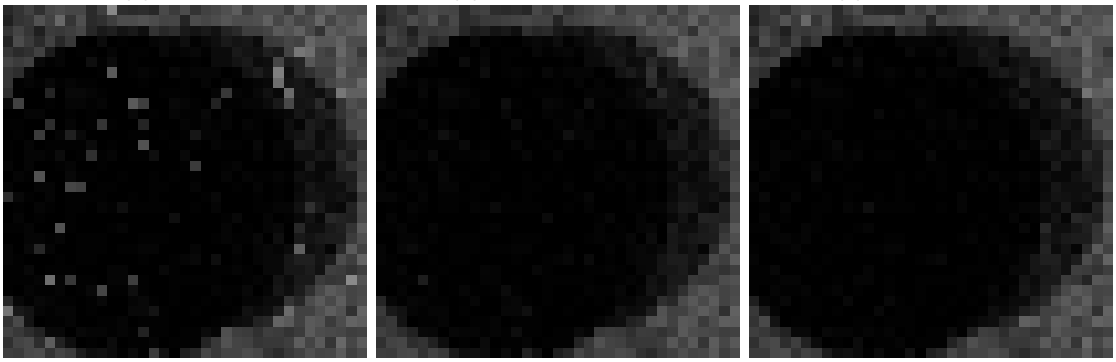
(a) 1000 Iterationen **(b)** Ground Truth

Abbildung A.4 Zu sehen sind die Streumatrizen der Projektion aus Datensatz I (Abbildung A.3), welche sich aus dem neuronalen Netz (a) und der Simulation (b) ergeben. Beide Bilder sind zum Maximum der Ground Truth (b) skaliert. Die Darstellung verwendet die invertierten Grauwerte, um einen größeren Kontrast zu bieten. Des Weiteren wird mit $\gamma = 0.1$ eine Gamma-Korrektur vorgenommen.

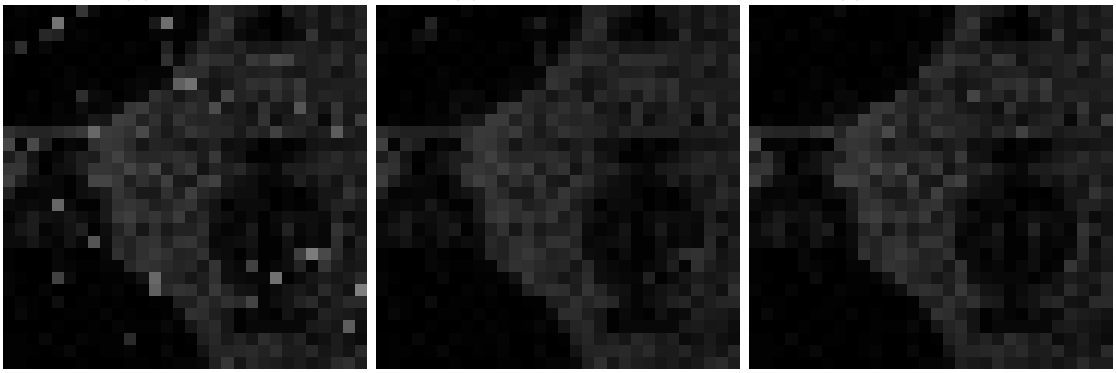
Datensatz II - Testdaten



(a) Original **(b)** 1000 Iterationen **(c)** Ground Truth

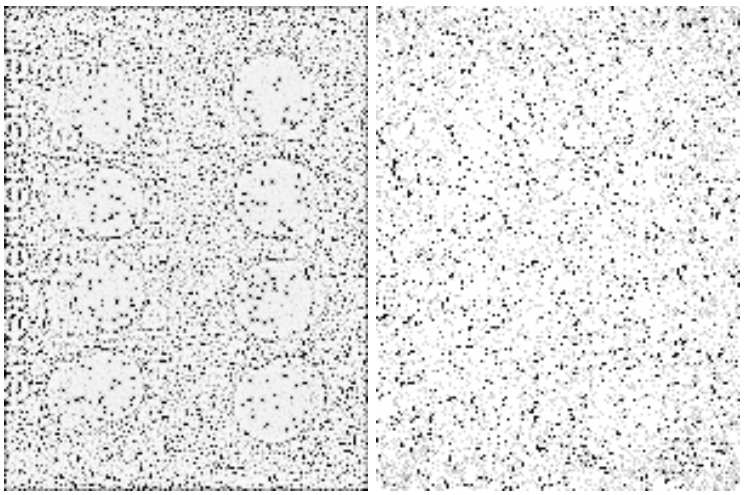


(d) Original **(e)** 1000 Iterationen **(f)** Ground Truth



(g) Original **(h)** 1000 Iterationen **(i)** Ground Truth

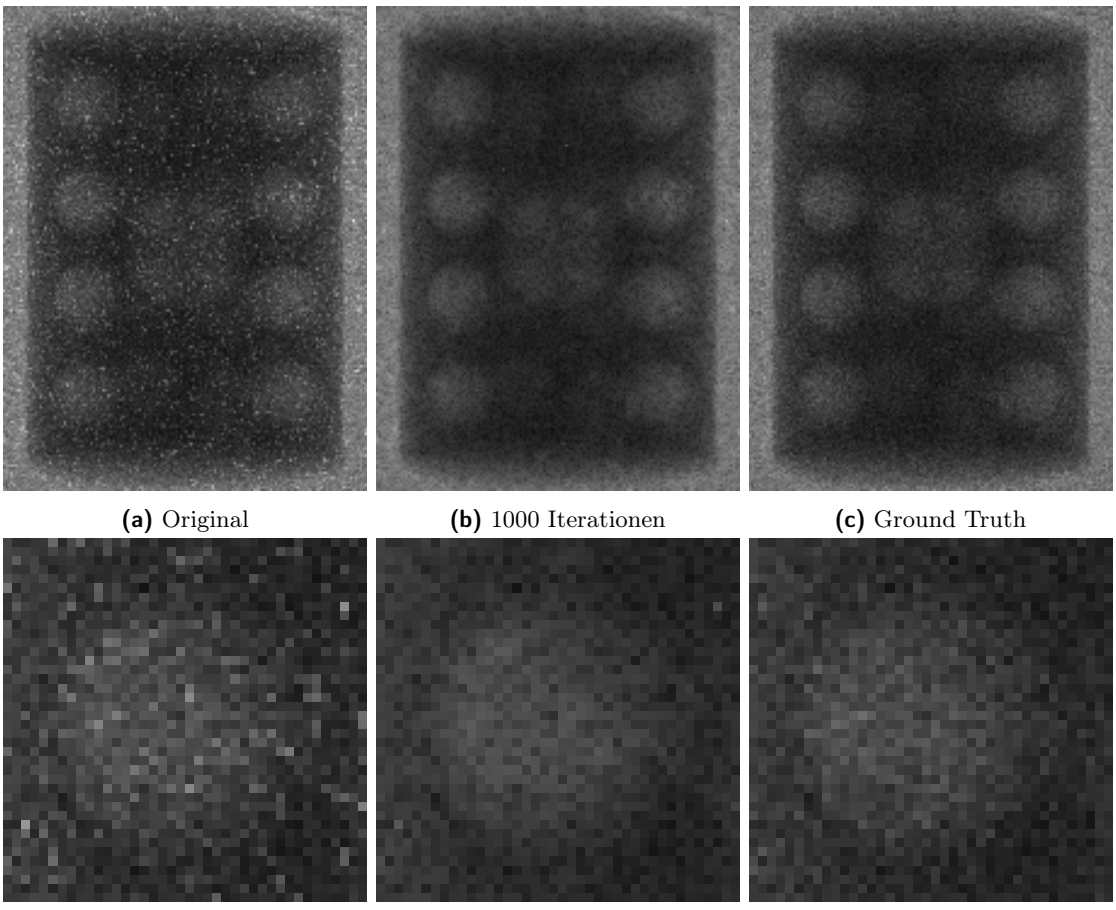
Abbildung A.5 Zu sehen ist eine Projektion aus den Testdaten des Datensatzes II. Die Abbildungen (a), (d) und (g) sind der unbearbeiteten Projektion entnommen. (b), (e) und (h) wurden durch das über 1000 Iterationen trainierte neuronale Netz gefiltert. (c), (f) und (i) wurden auf Basis der Simulationsdaten rauschreduziert.



(a) 1000 Iterationen **(b)** Ground Truth

Abbildung A.6 Zu sehen sind die Streumatrizen der Projektion aus Datensatz II (Abbildung A.5), welche sich aus dem neuronalen Netz (a) und der Simulation (b) ergeben. Beide Bilder sind zum Maximum der Ground Truth (b) skaliert. Die Darstellung verwendet die invertierten Grauwerte, um einen größeren Kontrast zu bieten. Des Weiteren wird mit $\gamma = 0.1$ eine Gamma-Korrektur vorgenommen.

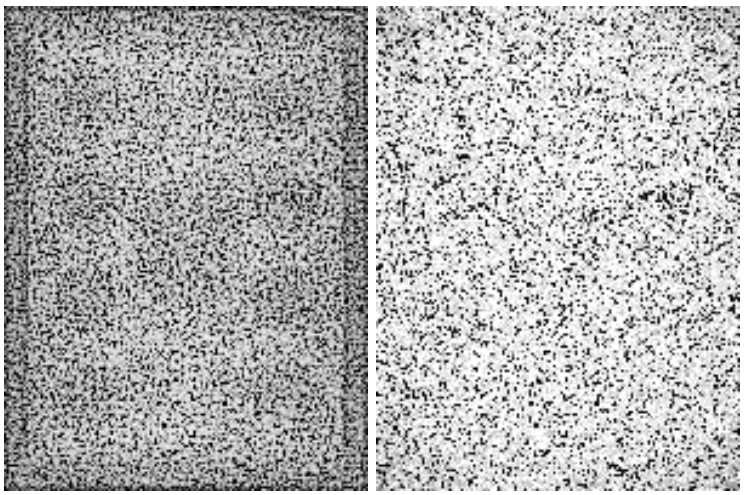
Datensatz IV - Testdaten



(a) Original **(b)** 1000 Iterationen **(c)** Ground Truth

(d) Original **(e)** 1000 Iterationen **(f)** Ground Truth

Abbildung A.7 Zu sehen ist eine Projektion aus den Testdaten des Datensatzes II. Die Abbildungen (a) und (d) sind der unbearbeiteten Projektion entnommen. (b) und (e) wurden durch das über 1000 Iterationen trainierte neuronale Netz gefiltert. (c) und (f) wurden auf Basis der Simulationsdaten rauschreduziert.

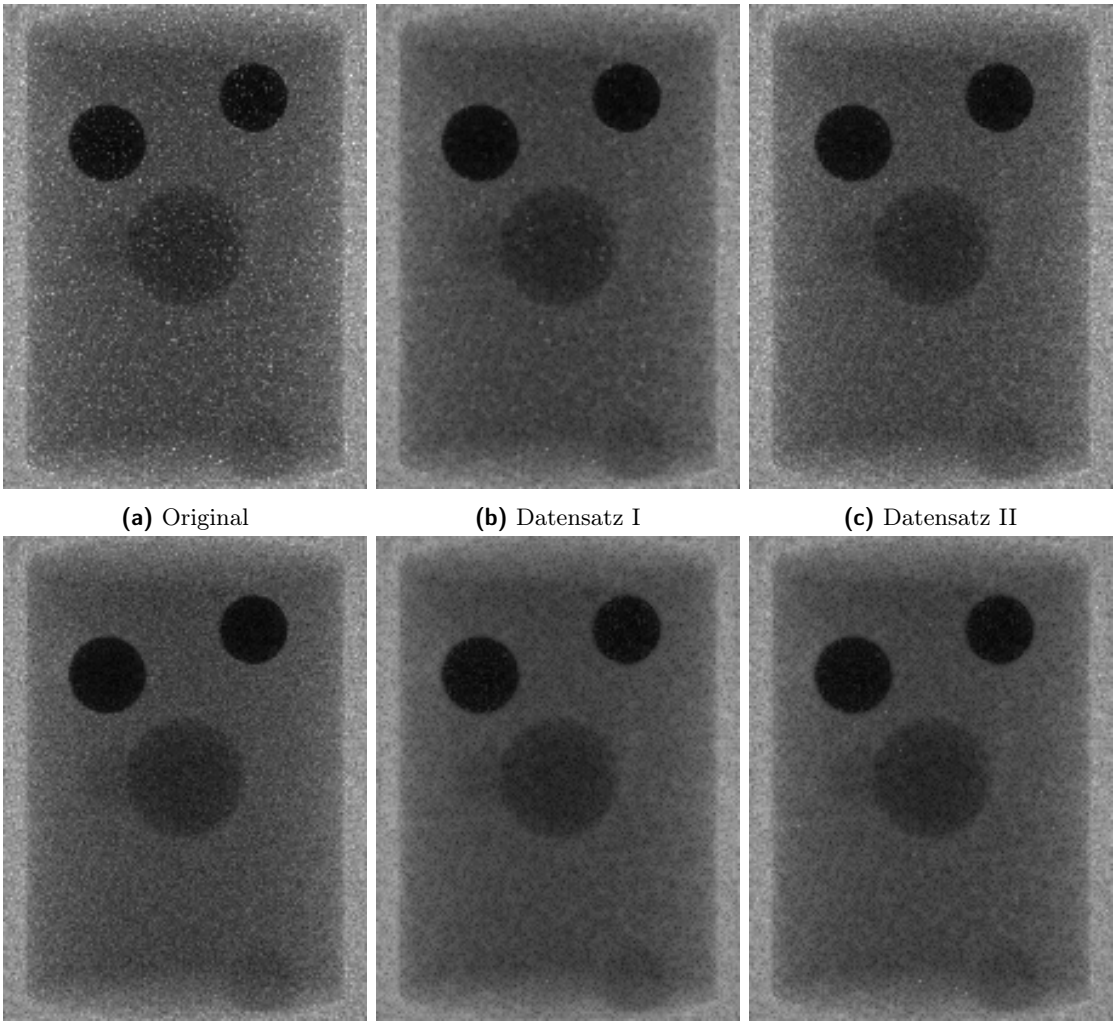


(a) 1000 Iterationen **(b)** Ground Truth

Abbildung A.8 Zu sehen sind die Streumatrizen der Projektion aus Datensatz II (Abbildung A.7), welche sich aus dem neuronalen Netz (a) und der Simulation (b) ergeben. Beide Bilder sind zum Maximum der Ground Truth (b) skaliert. Die Darstellung verwendet die invertierten Grauwerte, um einen größeren Kontrast zu bieten. Des Weiteren wird mit $\gamma = 0.1$ eine Gamma-Korrektur vorgenommen.

A.1.3 Validierungsdaten

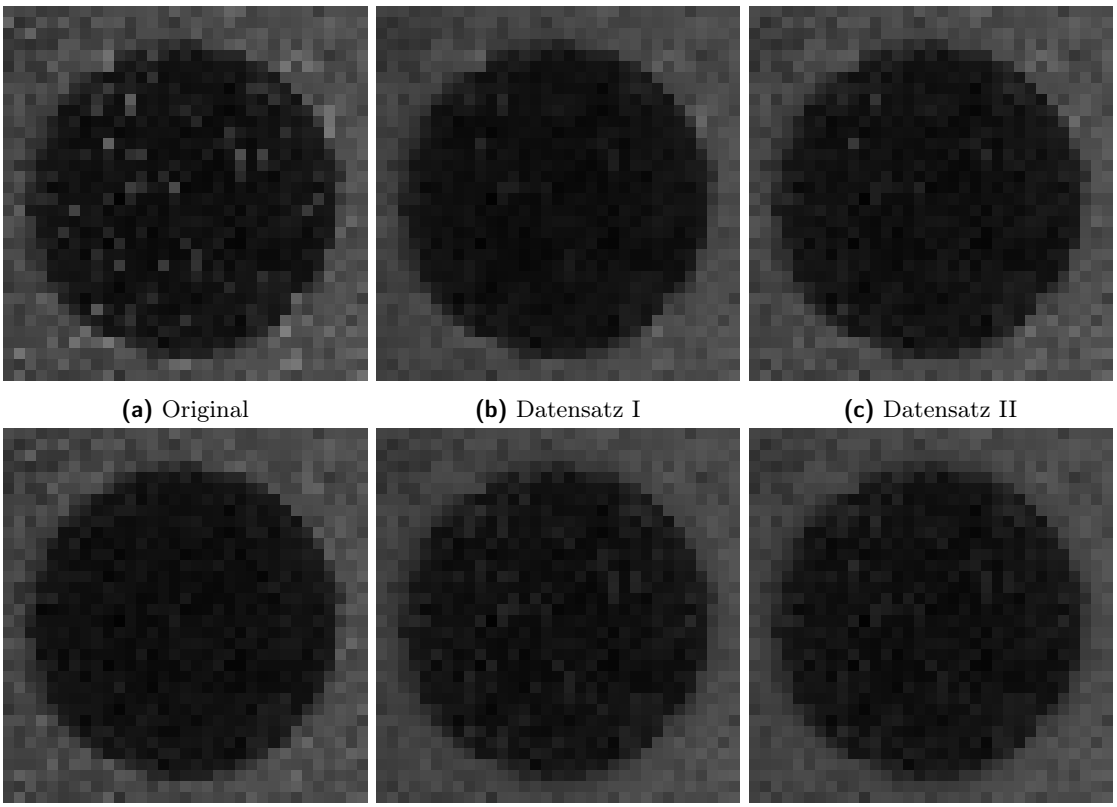
Validierungsphantom (c)



(a) Original **(b)** Datensatz I **(c)** Datensatz II

(d) Ground Truth **(e)** Datensatz III **(f)** Datensatz IV

Abbildung A.9 Es wird die Projektion des Validierungsphantoms c im Original in Abbildung (a), nach den Datensätzen I, II, III und IV gefiltert in (b), (c), (e) und (f) und als Ground Truth in Abbildung (d) dargestellt.

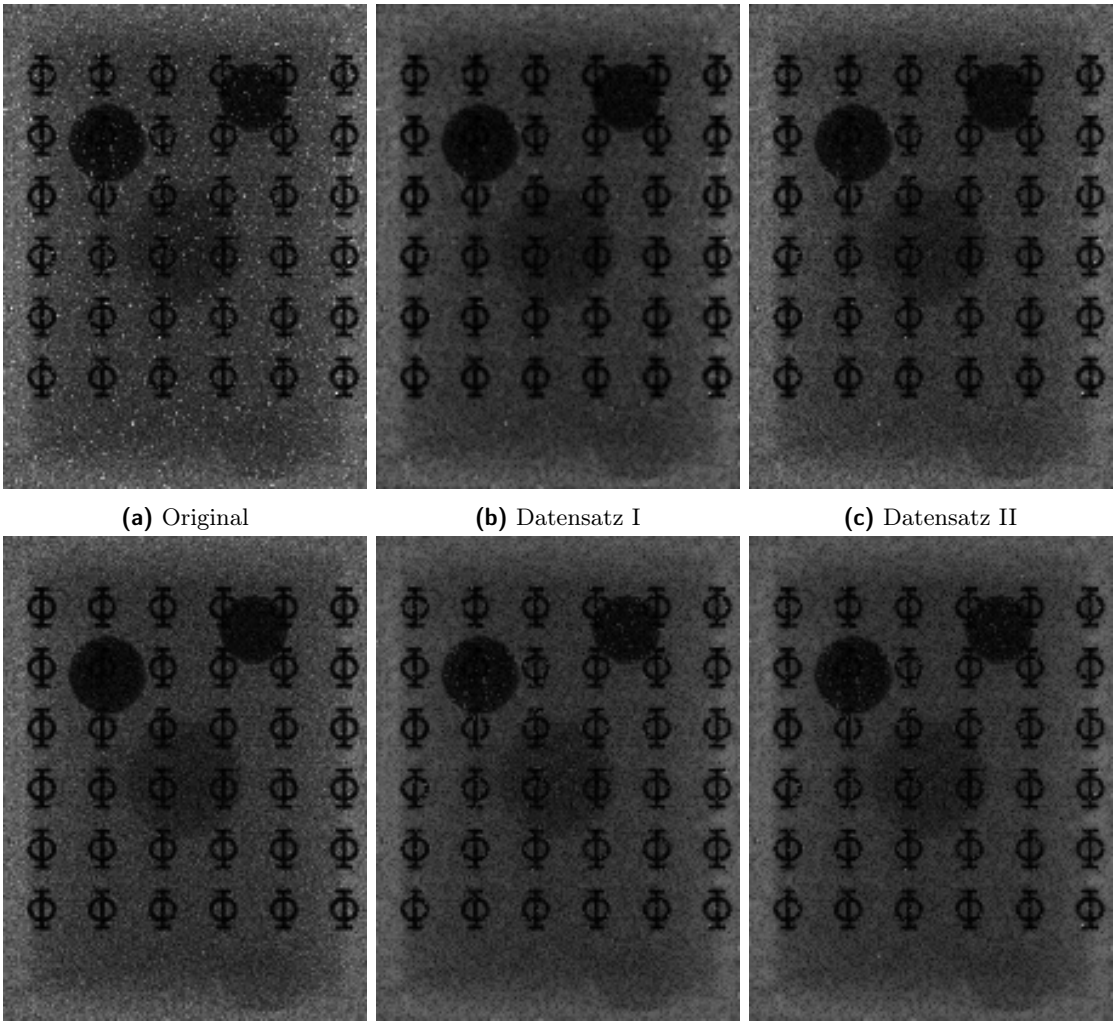


(a) Original **(b)** Datensatz I **(c)** Datensatz II

(d) Ground Truth **(e)** Datensatz III **(f)** Datensatz IV

Abbildung A.10 Es wird ein Ausschnitt des Validierungsphantoms c im Original in Abbildung (a), nach den Datensätzen I, II, III und IV gefiltert in (b), (c), (e) und (f) und als Ground Truth in Abbildung (d) dargestellt.

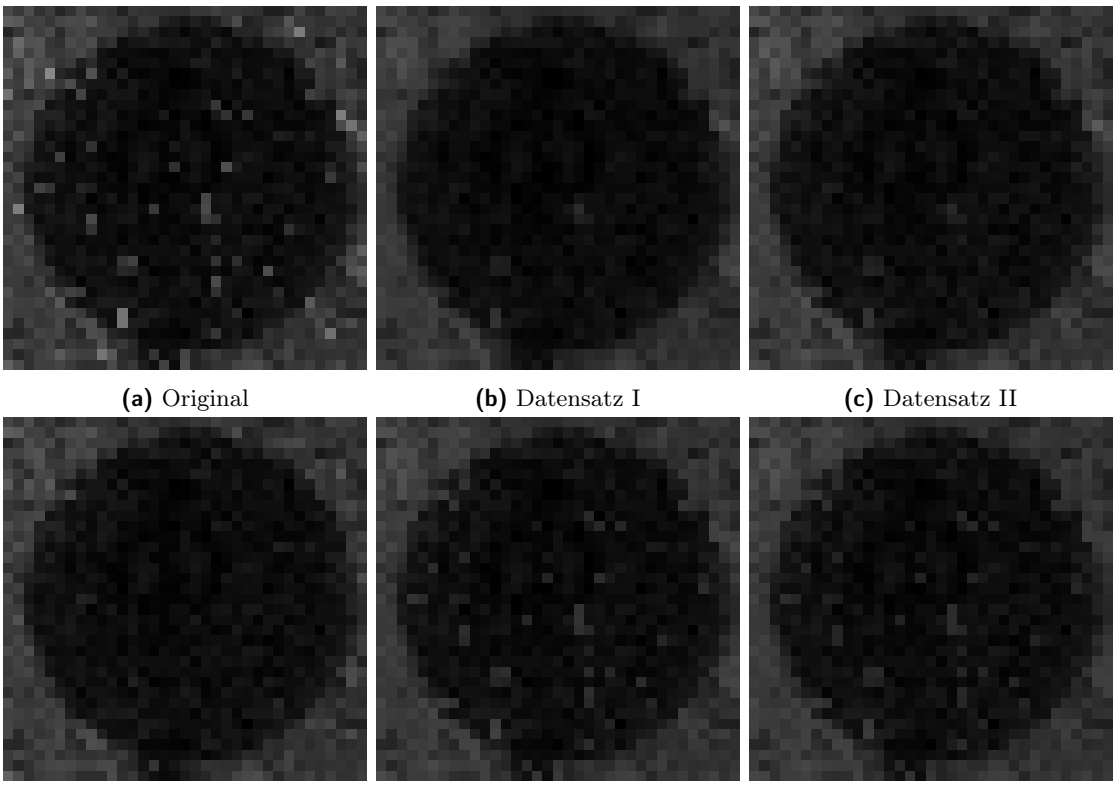
Validierungphantom (e)



(a) Original **(b)** Datensatz I **(c)** Datensatz II

(d) Ground Truth **(e)** Datensatz III **(f)** Datensatz IV

Abbildung A.11 Es wird die Projektion des Validierungsphantoms e im Original in Abbildung (a), nach den Datensätzen I, II, III und IV gefiltert in (b), (c), (e) und (f) und als Ground Truth in Abbildung (d) dargestellt.



(a) Original **(b)** Datensatz I **(c)** Datensatz II

(d) Ground Truth **(e)** Datensatz III **(f)** Datensatz IV

Abbildung A.12 Es wird ein Ausschnitt des Validierungsphantoms e im Original in Abbildung (a), nach den Datensätzen I, II, III und IV gefiltert in (b), (c), (e) und (f) und als Ground Truth in Abbildung (d) dargestellt.

A.2 python-Skripte

A.2.1 Programm zur Einlese der Projektionen aus einer Binärdatei

readprojection.py

```
import numpy as np
from matplotlib import pyplot as plt
def read_image(file, columns, rows):
  data = np.fromfile(file, '<f4')  # little-endian float32
  data.shape = (columns, rows)
  return data


columns = 150
rows = 200
inputdir = '/media/Data/Data_sjung/simulations/thirdCT_10000000_test4/'
#outputdir = '/media/Data/Data_sjung/thirdCT_10000000_1/'

rootpath = inputdir + "ct_results.root"
Branchname = "Hits"
projectionpath = inputdir + "ct_projection_000.dat"
projectionpath1 = inputdir + "ct_projection_001.dat"

projection = read_image(projectionpath, rows, columns)
plt.imshow(projection, cmap='viridis')
plt.show()
projection = read_image(projectionpath1, rows, columns)
plt.imshow(projection, cmap='viridis')
plt.show()

#projection_fft = np.fft.fft2(projection)
#projection_fft_abs = np.abs(projection_fft)
#projection_fft_abs_log = np.log(projection_fft_abs)
#plt.imshow(projection_fft_abs_log, cmap = 'viridis')
#plt.show()
```

A.2.2 Programm zur Umwandlung der ROOT-Dateien in python-arrays

pythonrootv2.py

```
import ROOT
import numpy as np
from matrix_class import data_matrices
import os, sys

####### DEFINE INPUT/OUTPUT PARAMETERS #######
columns = 150
rows = 200
base_inputdir = '/run/media/sj/SJ1/Bachelor/Data_sjung/simulations_5/
                                        thirdCT_400000000_120slices_'
numinputdirs = 50
inputdirs = []
```

```
base_base_outputdir = '/run/media/sj/SJ1/Bachelor/Data_sjung/simulations_5/
                                        thirdCT_400000000_120slices_'
base_outputdir = '/run/media/sj/SJ1/Bachelor/Data_sjung/simulations_5/
                                        thirdCT_400000000_120slices_10/arrays'
numoutputdirs = 120

for j in range(0,numinputdirs):   #make input directories
  inputdirs.append(base_inputdir + str(j)+'/')

for inputdir in inputdirs:
  outputdirs = []
  for j in range(0, numoutputdirs):   #make output directories
    outputdirs.append(inputdir + 'arrays' + str(j)+'/')
    if not os.path.exists(outputdirs[j]):
      os.mkdir(outputdirs[j], 0755)
  print inputdir

###### READ ROOT PATH #######
  rootpath = inputdir + "ct_results.root"
  Treename = "Hits"
  Treefile = ROOT.TFile(rootpath)
  Tree = Treefile.Get(Treename)
  n_Tree = Tree.GetEntries()

  time = numoutputdirs    #the number of time slices in the simulation
  timeslices = np.arange(0,time+1)
  numfeatures = 1   #the number of features to extract
  trackLength = 0   #define the feature to extract
  features = [[data_matrices(columns*rows, columns, rows) for _ in range(0,
                                        numfeatures)] for _ in range(0,time)]
  featurestrings = ['trackLength']
  print n_Tree
  j = 0

####### EXTRACT THE SELECTED DATA FROM THE ROOT TREE #######
  for i in range(int(n_Tree/1)):
    Tree.GetEntry(i)    #read the  ith Entry in the root tree
    if (Tree.time > j+1) & (Tree.time < j+2) & (Tree.time < time):    #discriminate
                                        the timeslice the entry belongs to. last
                                        condition is because seldom there are one
                                        or two counts counted after the actual
                                        simulation time
      j += 1
      features[j][trackLength].update(Tree, i, Tree.trackLength)
    elif (Tree.time > j) & (Tree.time < j+1):
      features[j][trackLength].update(Tree, i, Tree.trackLength)
    else:
      print('ERROR - Unknown Time: ', Tree.time)
####### MAKE DIFFERENT FORMATS OF THE DATA AND SAVE THE SELECTED DATA #######
  for k in range(0, time):
    print outputdirs[k]
    for l in range(0, numfeatures):
      features[k][l].reshape(columns, rows)
```

```
        features[k][l].makeprojection(columns, rows)
58      features[k][l].savearrays(featurestrings[l], outputdirs[k])
```

A.2.3 Klasse zur Verarbeitung und Speicherung der Simulationsdaten

matrix_class.py

```
0  import ROOT
   import numpy as np
2  from matplotlib import pyplot as plt

4  class data_matrices(object):
     """

6
     Parameters
8    ----------
     pixels : int
10     number of pixels = columns*rows
     columns : int
12     number of columns in a projection
     rows : int
14     number of rows in a projection
     IDsPerPixel : array, shape = [rows*columns]
16     array containing the number of events in each pixel
     vector : array, shape = [n*pixels]
18     the events, mapped to each pixel, n vectors appended to avoid double counting in
                                          one pixel
     matrix : array, shape = [n, rows, columns]
20     the vectors, reshaped to a three dimensional matrix, which has the detector
                                         dimensions as two dimensions and the
                                         events per pixel in the third dimension
     IDsPerPixel_matrix : array, shape = [rows, columns]
22     a matrix of the detector dimensions, whose elements are the number of counted
                                         events in the corresponding pixel
     hist : the histogram showing the distribution of the values in all pixels
24   """
     def __init__(self, pixels, columns, rows, vector = [], matrix = [], IDsPerPixel =
                                         [], IDsPerPixel_matrix = []):
26     self.pixels = columns*rows
       self.columns = columns
28     self.rows = rows

30     self.IDsPerPixel = np.zeros((pixels))
       self.vector = np.zeros((pixels))
32
       self.IDsPerPixel_matrix = np.zeros((rows, columns))
34     self.matrix = np.zeros((1, rows, columns))
       self.IDsPerPixel_fft_matrix = np.zeros((rows, columns))
36     self.hist = []

38   def update(self, Tree, i, Tree_object):
       """Get an entry from the ROOT-Tree and log it to the corresponding pixel in the
                                         vector. Increment the IDsPerPixel counter
```

```
                                        and introduce new layer if necessary
40
    Parameters
42  ----------
    Tree : ROOT-TFile
44    The Tree containing the branches
    i : int
46    the iterator, specifiying which element to read the branches from
    Tree_object : KORRR
48    The selected branch

50  Returns
    -------
52  self
    """
54  Tree.GetEntry(i)
    newlayer = np.zeros((self.pixels))
56  self.vector[int(Tree.pixelID + self.IDsPerPixel[int(Tree.pixelID)]*self.pixels)]
                                          = Tree_object
    self.IDsPerPixel[int(Tree.pixelID)] +=1
58  if (self.IDsPerPixel[int(Tree.pixelID)] * self.pixels >= len(self.vector)):
      self.vector = np.append(self.vector, newlayer)
60
  def reshape(self, columns, rows):
62  """Reshape the one-dimensional vector to the three-dimensional matrix.

64  Parameters
    ----------
66  columns : int
      number of columns in a projection
68  rows : int
      number of rows in a projection
70
    Returns
72  -------
    self
74  """
    var = len(self.vector)/(columns*rows)
76  self.matrix = np.reshape(self.vector, (int(var), rows, columns))

78  def makeprojection(self, columns, rows):
    """Count the IDs referring to the same pixel and write them in
                                          IDsPerPixel_matrix
80
    Parameters
82  ----------
    columns : int
84    number of columns in a projection
    rows : int
86    number of rows in a projection

88  Returns
    -------
```

```
90      self
        """
92      self.IDsPerPixel_matrix = np.swapaxes(np.array([[len(self.matrix[:,i,j][self.
                                        matrix[:,i,j] != 0]) for i in range(0,rows
                                        )] for j in range(0,columns)]), 0,1)

94    def makehist(self):
        """reproduce the root-histogram
96      """
        nums = self.matrix.shape[0]*self.matrix.shape[1]*self.matrix.shape[2]
98      custom_row_matrix = np.reshape(self.matrix, nums)
        matrixwithoutzeros  = custom_row_matrix[custom_row_matrix!=0]
100     self.hist = np.histogram(matrixwithoutzeros, 100)

102   def plot(self, name, directory):
        """plot the projection and the histogram
104
        Parameters
106     ----------
        name : str
108       the name of the object
        directory : str
110       the directory to save the plots
        """
112     plt.imshow(self.IDsPerPixel_matrix, cmap = 'gray')
        plt.title(name)
114     plt.savefig(directory + name + '.tif')
        plt.close()
116     x = self.hist[1][1:len(self.hist[1])-1]
        y = self.hist[0][1:]
118     plt.plot(x, y, '.')
        plt.ylim([0, int(max(y))])
120     plt.title(name + '_histogram')
        plt.savefig(directory + name + '_histogram' + '.tif')
122     plt.close()

124   def getfft(self):
        """make a fourier transform of the IDsPerPixel matrix
126     """
        nozeros = np.abs(np.fft.fft2(self.IDsPerPixel_matrix))
128     nozeros[nozeros == 0] = 1
        self.IDsPerPixel_fft_matrix = np.log(nozeros)
130
      def getfftshift(self):
132     """make fourier transform with the low frequencies in the center of the image
        """
134     nozeros = np.abs(np.fft.fftshift(np.fft.fft2(self.IDsPerPixel_matrix)))
        nozeros[nozeros == 0] = 1
136     self.IDsPerPixel_fft_matrix = np.log(nozeros)

138   def getifft(self):
        """make an inverse fourier transform of the fourier transformed IDsPerPixel
                                        matrix
```

```
140     """
        self.IDsPerPixel_fft_matrix = np.abs(np.fft.ifft2(np.exp(self.
                                            IDsPerPixel_fft_matrix)))
142
    def getifftshift(self):
144     """make an inverse fourier transform of the shifted fourier transformed
                                            IDsPerPixel matrix
        """
146     self.IDsPerPixel_fft_matrix = np.abs(np.fft.ifft2(np.fft.ifftshift(np.exp(self.
                                            IDsPerPixel_fft_matrix))))

148 def savearrays(self, name, directory):
        """save the numpy arrays in the directory
150
        Parameters
152     ----------
        name : str
154       the name of the object
        directory : str
156       the directory to save the arrays
        """
158     np.save(directory + name + '_projection' + '.npy', self.IDsPerPixel_matrix)
        np.save(directory + name + '_matrix' + '.npy', self.matrix)
160     np.save(directory + name + '_projectionvector' + '.npy', self.IDsPerPixel)
        np.save(directory + name + '_matrixvector' + '.npy', self.vector)
162
    def loadarrays(self, name, inputdir):
164     """load the numpy arrays from the directory

166     Parameters
        ----------
168     name : str
          the name of the object
170     inputdir : str
          the directory to load the arrays from
172     """
        self.IDsPerPixel_matrix = np.load(inputdir + name + '_projection' + '.npy')
174     self.matrix = np.load(inputdir + name + '_matrix' + '.npy')
        self.IDsPerPixel = np.load(inputdir + name + '_projectionvector' + '.npy')
176     self.vector = np.load(inputdir + name + '_matrixvector' + '.npy')

178 def loadpicture(self, name, inputdir):
        """load the projection-array from the directory
180
        Parameters
182     ----------
        name : str
184       the name of the object
        inputdir : str
186       the directory to load the projection-array from
        """
188     self.IDsPerPixel_matrix = np.load(inputdir + name + '_projection' + '.npy')
```

```
190   def filterminmax(self, toFilter, minvalue, maxvalue):
        """filter the data by a mininmal and a maximal value
192
        Parameters
194     ----------
        toFilter : data_matrices object
196       the object to filter and then write to self
        minvalue : float
198       the minimal value for the data in the toFilter object
        maxvalue : float
200       the maximal value for the data in the toFilter object
        """
202     self.vector = np.array(toFilter.vector)
        self.vector[(self.vector <= minvalue)] = 0
204     self.vector[(self.vector >= maxvalue)] = 0
        self.reshape(self.columns, self.rows)
206     self.makeprojection(self.columns, self.rows)

208   def joinarrays(self, toJoin):
        """join the array sets of two objects
210
        Parameters
212     ----------
        toJoin : data_matrices object
214       the object to join to self
        """
216     self.IDsPerPixel_matrix += toJoin.IDsPerPixel_matrix
        self.matrix = np.append(self.matrix, toJoin.matrix, axis = 0)
218     self.IDsPerPixel += toJoin.IDsPerPixel
        self.vector = np.append(self.vector, toJoin.vector, axis = 0)
```

A.2.4 Skript zum Zusammenfügen parallelisierter Simulationen

adder.py

```
0   import ROOT
    import numpy as np
2   from matrix_class import data_matrices
    import os, sys
4
    ####### DEFINE INPUT/OUTPUT PARAMETERS #######
6   columns = 150
    rows = 200
8   base_inputdir_1 = '/run/media/sj/SJ1/Bachelor/Data_sjung/simulations_4/
                                          thirdCT_400000000_120slices_'
    base_inputdir_2 = '/arrays'
10  numinputdirs_1 = 50
    numinputdirs_2 = 120
12  inputdirs = []
    base_outputdir = '/run/media/sj/Seagate Expansion Drive/Data_sjung/cumulated_4/
                                          thirdCT_400000000_120slices_cumulated'
14  outputdirs = []
```

```
for i in range(0,numinputdirs_2):   #make input/output directories
  metainputdirs = []
  for j in range(0,numinputdirs_1):
    metainputdirs.append(base_inputdir_1 + str(j) + base_inputdir_2 + str(i) +'/')
  inputdirs.append(metainputdirs)
  outputdirs.append(base_outputdir + str(i) + '/')
  if not os.path.exists(outputdirs[-1]):
      os.mkdir(outputdirs[-1], 0755)

####### READ AND CUMULATE THE DATA #######
for k in range(0, numinputdirs_2):
  numfeatures = 1   #the number of features to cumulate
  features = [data_matrices(columns*rows, columns, rows) for _ in range(0,
                                        numfeatures)]   #empty data matrices
  features_temp = [data_matrices(columns*rows, columns, rows) for _ in range(0,
                                        numfeatures)]    #empty data matrices
  featurestrings = ['trackLength']

  for i in range(0, numinputdirs_1):
    print(inputdirs[k][i])
    for j in range(0, numfeatures):
      features_temp[j].loadarrays(featurestrings[j], inputdirs[k][i])   #load the
                                        single data arrays
      features[j].joinarrays(features_temp[j])    #cumulate by joining the new data
                                        arrays to the existing ones
    print('sum(features_temp[i][j].IDsPerPixel)', sum(features_temp[j].IDsPerPixel))
    print('sum(features[i][j].IDsPerPixel)', sum(features[j].IDsPerPixel))

  print(outputdirs[k])
  for j in range(0, numfeatures):
    features[j].savearrays(featurestrings[j], outputdirs[k])    #save the cumulated
                                        arrays
```

A.2.5 Skript zum Filtern durch den TrackLength-Parameter

filter_by_trackLength.py

```
import numpy as np
import matplotlib.pyplot as plt

from matrix_class import data_matrices
from convolve_class import convolveWork
from neuralnet_class import NetWork

def make_trackLength_filtered(columns, rows, numinputdirs, base_inputdir):
  """read the tracklength data and filter it by the geometry and save the resulting
                                        matrices

  Parameters
  ----------
  columns : int
    number of columns
  rows : int
```

```
    number of rows
  numinputdirs : int
    number of images to load
  base_inputdir : str
    path to the images

  Returns
  -------
  trackLength : list, shape = [numinputdirs]
    trackLength[i] : data_matrices
    each element includes the projection
  trackLengthwindow : list, shape = [numinputdirs]
    trackLengthwindow[i] : data_matrices
    each element includes the projection, windowed by the trackLength
  """

  ####### GEOMETRY #######
  focal_spot = .025   #size of the focal spot
  FDD = 500.    #Focus-Detector-Distance
  FOD = 100.    #Focus-Object-Distance
  ODD = FDD - FOD   #Objeckt-Detector-Distance
  Oxwidth = 13.   #Object-width in x-direction
  Oywidth  = 35.    #Object-width in y-direction
  lowfactor = 0.99    #decrease the lower bound a little bit
  highfactor = 1.014 #anpassen! #increase the higher bound a little bit
  longest_path = highfactor * np.sqrt(FDD**2 + (FDD/(FOD-Oxwidth/2)*Oywidth/2)**2)
                                        #calculate geometric longest path
  shortest_path = lowfactor * FDD   #shortest path

  trackLength = [data_matrices(columns*rows, columns, rows) for _ in range(0,
                                        numinputdirs)]   #initialize lists
  trackLengthwindow = [data_matrices(columns*rows, columns, rows) for _ in range(0,
                                        numinputdirs)]

  for j in range(0,numinputdirs):
    outputdir = base_inputdir + str(j) + '/'    #set inputdirectory
    inputdir = base_inputdir + str(j) + '/'
    print(inputdir)
    trackLength = data_matrices(columns*rows, columns, rows)
    trackLengthwindow = data_matrices(columns*rows, columns, rows)
    trackLength.loadarrays('trackLength', inputdir)   #load the data
    trackLengthwindow.filterminmax(trackLength, shortest_path, longest_path)    #
                                        filter the data by the tracklength
    trackLengthwindow.savearrays('trackLengthwindow', inputdir)   #save the filtered
                                        data
#   trackLength[j].loadarrays('trackLength', inputdir)    #load the data
#   trackLengthwindow[j].filterminmax(trackLength[j], shortest_path, longest_path)
                                        #filter the data by the tracklength
#   trackLengthwindow[j].savearrays('trackLengthwindow', inputdir)    #save the
                                        filtered data
    #plt.subplot(121)
    #plt.imshow(trackLength[j].IDsPerPixel_matrix, cmap = 'gray')
    #plt.subplot(122)
```

```
    #plt.imshow(trackLengthwindow[j].IDsPerPixel_matrix, cmap = 'gray')
    #plt.show()
  return trackLength, trackLengthwindow

columns = 150
rows = 200

base_inputdir = '/run/media/sj/Seagate Expansion Drive/Data_sjung/cumulated_5/
                                        thirdCT_400000000_120slices_cumulated'
numinputdirs = 120

a,b = make_trackLength_filtered(columns, rows, numinputdirs, base_inputdir)
```

A.2.6 Das Hauptprogramm zur Merkmalerstellung und zum trainieren des neuronalen Netzes

seventh.py

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from PIL import Image
import os, sys

from matrix_class import data_matrices
from convolve_class import convolveWork
from neuralnet_class import NetWork

def load_trackLength_filtered(columns, rows, numinputdirs, base_inputdir):
  """load the trackLength and trackLenthwindow projections, meaning the unfiltered
                                        and filtered data

  Parameters
  ----------
  columns : int
    number of columns
  rows : int
    number of rows
  numinputdirs : int
    number of images to load
  base_inputdir : str
    path to the images

  Returns
  -------
  trackLength : list, shape = [numinputdirs]
    trackLength[i] : data_matrices
    each element includes the projection
  trackLengthwindow : list, shape = [numinputdirs]
    trackLengthwindow[i] : data_matrices
    each element includes the projection, windowed by the trackLength
  """
```

```
  trackLength = [data_matrices(columns*rows, columns, rows) for _ in range(0,
                                          numinputdirs)]   #initialize lists
  trackLengthwindow = [data_matrices(columns*rows, columns, rows) for _ in range(0,
                                          numinputdirs)]
  for j in range(0, numinputdirs):
    inputdir = base_inputdir + str(j)+'/'   #set inputdirectory
    print(inputdir)
    trackLength[j].loadpicture('trackLength', inputdir)   #load the data
    trackLengthwindow[j].loadpicture('trackLengthwindow', inputdir)   #load the data
  return trackLength, trackLengthwindow

def min_max_scale(image, data_range = 'uint16'):
  """scale the image with its max value to the maximum of the data type

  Parameters
  ----------
  image : array, shape = [columns, rows]
    the image to scale
  data_range : string
    the name of the data type

  Returns
  -------
  image.astype(data_range) : array, shape = [columns, rows]
    the scaled image
  """
  if np.max(image) > 0:
    image*=(np.iinfo(data_range).max/np.max(image))
  return image.astype(data_range)

def min_custommax_scale(image, custommax, data_range = 'uint16'):
  """scale the image with to the the data type considering the given maximum

  Parameters
  ----------
  image : array, shape = [columns, rows]
    the image to scale
  custommax : float
    the Value that should be assigned to maximal available value of the data type
  data_range : string
    the name of the data type

  Returns
  -------
  image.astype(data_range) : array, shape = [columns, rows]
    the scaled image
  """
  if custommax > 0:
    image*=(np.iinfo(data_range).max/custommax)
  return image.astype(data_range)

columns = 150
rows = 200
```

```
84
dataset = 1 #1,2,3,4
86 if ((dataset == 2 )):
    datanums = 1
88   numinputdirs = 120
    num_trainsets = 100
90   num_testsets = 20
elif ((dataset == 4 )|( dataset == 3)):
92   datanums = 4
    numinputdirs = 120
94   num_trainsets = 100
    num_testsets = 20
96 elif (dataset == 1):
    datanums = 1
98   numinputdirs = 30
    num_trainsets = 25
100  num_testsets = 5
base_inputdir = '/run/media/sj/Seagate Expansion Drive/Data_sjung/cumulated_' + str(
                                        dataset+1) + '/thirdCT_' + str(datanums) +
                                         '00000000_' + str(numinputdirs) + '
                                        slices_cumulated'
102

selected_data_range = 'uint16'
104

trackLength, trackLengthwindow = load_trackLength_filtered(columns, rows,
                                        numinputdirs, base_inputdir)    #read the
                                        test and training data
106

####### make the kernels for the convolutional features #######
108 edge_1 = np.array([[1,0,-1],[0,0,0],[-1,0,1]])
edge_2 = np.array([[0,1,0],[1,-4,1],[0,1,0]])
110 edge_3 = np.array([[-1,-1,-1],[-1,8,-1],[-1,-1,-1,]])
sharp_1 = np.array([[0,-1,0],[-1,5,-1],[0,-1,0]])
112 kernels = [np.ones((3,3)), np.ones((5,5)), edge_1, edge_2, edge_3, sharp_1]
n_features_est = len(kernels)+4
114

####### LOAD & PREPARE TRAINDATA #######
116 X_first = np.empty((num_trainsets, rows, columns))    #initialize original
                                        projection array
y_first = np.empty((num_trainsets, rows, columns))    #initialize desired projection
                                         array
118 X_train = np.empty((num_trainsets, rows*columns, n_features_est+1))   #initialize
                                        the training data arrays
y_train = np.empty((num_trainsets, rows*columns))
120

makefeatures_train = []
122 for i in range(0,num_trainsets):
    X_first[i] = trackLength[i].IDsPerPixel_matrix    #load the original projections
124  y_first[i] = X_first[i] - trackLengthwindow[i].IDsPerPixel_matrix   #load the
                                        filtered, desired projections
    makefeatures_train.append(convolveWork(X_first[i], y_first[i]))   #initialize the
                                        makefeatures object to make the features
126  makefeatures_train[i].features = [makefeatures_train[i].maxpool(),
```

```
                                        makefeatures_train[i].minpool(),
                                        makefeatures_train[i].meanpool(),
                                        makefeatures_train[i].stdpool()]    #
                                        append the standart features
  for j in range(len(kernels)):
128   makefeatures_train[i].features.append(makefeatures_train[i].conv_w_kernel(
                                        kernels[j]))    #append the convolutional
                                        features
  n_features = len(makefeatures_train[i].features)    #number of features
130   if n_features != n_features_est:
    print('ERROR:WRONG n_feature INITIALIZATION')
132   X_train[i,:,:], y_train[i,:], X_train_rescale, y_train_rescale= makefeatures_train
                                        [i].vectorize(set_scale = True)   #get the
                                         vectorized training data for the NetWork

134 ####### LOAD & PREPARE TESTDATA #######
X_first = np.empty((num_testsets, rows, columns))   #initialize original projection
                                        array
136 y_first = np.empty((num_testsets, rows, columns))   #initialize desired projection
                                        array
X_test = np.empty((num_testsets, rows*columns, n_features+1))   #initialize the
                                        training data arrays
138 y_test = np.empty((num_testsets, rows*columns))

140 makefeatures_test = []
for i in range(0, num_testsets):
142   X_first[i] = trackLength[num_trainsets + i].IDsPerPixel_matrix    #load the
                                        original projections
  y_first[i] = X_first[i] - trackLengthwindow[num_trainsets + i].IDsPerPixel_matrix
                                          #load the filtered, desired projections
144   makefeatures_test.append(convolveWork(X_first[i], y_first[i]))    #initialize the
                                        makefeatures object to make the features
  makefeatures_test[i].features = [makefeatures_test[i].maxpool(), makefeatures_test
                                        [i].minpool(), makefeatures_test[i].
                                        meanpool(), makefeatures_test[i].stdpool()
                                        ]
146   for j in range(len(kernels)):
    makefeatures_test[i].features.append(makefeatures_test[i].conv_w_kernel(kernels[
                                        j]))
148   X_test[i,:,:], y_test[i,:], X_test_rescale, y_test_rescale = makefeatures_test[i].
                                        vectorize(set_scale=False, custom_scale =
                                        True, X_cuscale = X_train_rescale,
                                        y_cuscale = y_train_rescale)

150 ####### CALCULATE & OPTIMIZE WEIGHTS #######
n_cycles = 5      #how many cycles the NetWork will be trained
152 n_output_features = 1   #one means, that one pixel with multiple features in the
                                        input layer will result in one pixel in
                                        the output layer
n_hidden = np.arange(1,100, 1)    #n_features + 1
154 etas = np.arange(0.001, 0.1, 0.01)
batchsizes = np.arange(0,rows*columns,30)[1:]
156 fixed = np.array([1])   #select if no hyperparameter modulation is wanted
```

```
158 parameters = [n_hidden, etas, batchsizes, fixed]
    parameter_labels = ['n_hidden', 'etas', 'batchsizes', 'None']
160
    eta_index = 0
162 batchsize_index = 10
    hidden_unit_index = 10  # hier w"are anstatt einer zahl ein array 'beliebiger' form
                                          denkbar, um auch mehrere schichten zu
                                          ermoeglichen
164
    do_hyparmod = False
166
    indexes = [hidden_unit_index, eta_index, batchsize_index, 1]
168 if do_hyparmod:
      modpar = 1  #modulate parameter number  0:n_hidden, 1:etas, 2:batchsize, 3:no
                                          hyperparameter modulation
170   cost_train = np.empty(parameters[modpar].shape[0])
      cost_L2_train = np.empty(parameters[modpar].shape[0])
172   cost_test = np.empty(parameters[modpar].shape[0])
      cost_L2_test = np.empty(parameters[modpar].shape[0])
174 else:
      modpar = 3
176
    ####### ITERATE OVER HYPERPARAMETERS #######
178 for m in range(0, parameters[modpar].shape[0]):
      print('Iteration:', m, '/', parameters[modpar].shape[0])
180   indexes[modpar] = m
      print('layers:',np.array([n_features+1, parameters[0][indexes[0]],
                                          n_output_features]))
182   print('eta:', parameters[1][indexes[1]])
      print('cycles:', n_cycles)
184   print('batchsize:', parameters[2][indexes[2]])

186   Net1 = NetWork(hidden_units = np.array([n_features+1, parameters[0][indexes[0]],
                                          n_output_features]), eta = parameters[1][
                                          indexes[1]], cycles = n_cycles, batchsize
                                          = parameters[2][indexes[2]], shuffle =
                                          False, seed = 1)   #initialize the NetWork
      optimized_train = Net1.fit(X_train, y_train)    #fit the data and return the
                                          optimized pictures for the training data
188
      ####### CALCULATE RESULTS #######
190   X_train_out = np.empty((X_train.shape[0], rows, columns))   #initialize the output
                                           arrays
      y_train_out = np.empty((y_train.shape[0], rows, columns))
192   optimized_train_out = np.empty((y_train.shape[0], rows, columns))
      if do_hyparmod:
194     cost_train_temp = 0
        cost_L2_train_temp = 0
196   for j in range(0,X_train.shape[0]):   #make the output arrays from the network-
                                          output
        X_train_out[j,:,:], y_train_out[j,:,:], optimized_train_out[j,:,:] =
                                          makefeatures_train[j].rescale_and_reshape(
```

```
                                        X_train[j,:,0], y_train[j,:],
                                        optimized_train[j,:], rows, columns)
198     if do_hyparmod:
          cost_train_temp += Net1.cost_function(optimized_train, y_train[j,:])
200       cost_L2_train_temp += Net1.cost_L2(optimized_train, y_train[j,:])
    if do_hyparmod:
202   cost_train[m] = cost_train_temp/X_train.shape[0]
      cost_L2_train[m] = cost_L2_train_temp/X_train.shape[0]
204
    ####### CALCULATE RESULTS ########
206 X_test_out = np.empty((X_test.shape[0], rows, columns))
    y_test_out = np.empty((y_test.shape[0], rows, columns))
208 optimized_test_out = np.empty((y_test.shape[0], rows, columns))
    if do_hyparmod:
210   cost_test_temp = 0
      cost_L2_test_temp = 0
212 for j in range(0,X_test.shape[0]):
      optimized_test = Net1.predict(X_test[j,:,:])
214   X_test_out[j,:,:], y_test_out[j,:,:], optimized_test_out[j,:,:] =
                                        makefeatures_test[j].rescale_and_reshape(
                                        X_test[j,:,0], y_test[j,:], optimized_test
                                        , rows, columns)
      if do_hyparmod:
216     cost_test_temp += Net1.cost_function(optimized_test, y_test[j,:])
        cost_L2_test_temp += Net1.cost_L2(optimized_test, y_test[j,:])
218 if do_hyparmod:
      cost_test[m] = cost_test_temp/X_test.shape[0]
220   cost_L2_test[m] = cost_L2_test_temp/X_test.shape[0]

222 ####### PREPARE VALIDATION DATA #######
    valid_inputdir_1 = '/run/media/sj/Seagate Expansion Drive/Data_sjung/simulations_1/
                                        thirdCT_100000000_test'
224 valid_inputdir_2 = '/run/media/sj/Seagate Expansion Drive/Data_sjung/simulations_1/
                                        thirdCT_10000000_test'
    num_valid_1 = 8
226 num_valid_2 = 8

228 trackLength, trackLengthwindow = load_trackLength_filtered(columns, rows,
                                        num_valid_1, valid_inputdir_1)    #just
                                        read the data
    X_first = np.empty((num_valid_1, rows, columns))    #initialize original projection
                                        array
230 y_first = np.empty((num_valid_1, rows, columns))    #initialize desired projection
                                        array
    X_valid_1 = np.empty((num_valid_1, rows*columns, n_features+1))   #initialize the
                                        training data arrays
232 y_valid_1 = np.empty((num_valid_1, rows*columns))

234 makefeatures_valid_1 = []
    for i in range(0, num_valid_1):
236   X_first[i] = trackLength[i].IDsPerPixel_matrix    #load the original projections
      y_first[i] = X_first[i] - trackLengthwindow[i].IDsPerPixel_matrix   #load the
                                        filtered, desired projections
```

```
238   makefeatures_valid_1.append(convolveWork(X_first[i], y_first[i]))   #initialize
                                          the makefeatures object to make the
                                          features
      makefeatures_valid_1[i].features = [makefeatures_valid_1[i].maxpool(),
                                          makefeatures_valid_1[i].minpool(),
                                          makefeatures_valid_1[i].meanpool(),
                                          makefeatures_valid_1[i].stdpool()]
240   for j in range(len(kernels)):
        makefeatures_valid_1[i].features.append(makefeatures_valid_1[i].conv_w_kernel(
                                          kernels[j]))
242   X_valid_1[i,:,:], y_valid_1[i,:], X_valid_1_rescale, y_valid_1_rescale =
                                          makefeatures_valid_1[i].vectorize(
                                          set_scale=False, custom_scale = True,
                                          X_cuscale = X_train_rescale, y_cuscale =
                                          y_train_rescale)

244 ####### CALCULATE RESULTS #######
    X_valid_1_out = np.empty((X_valid_1.shape[0], rows, columns))
246 y_valid_1_out = np.empty((y_valid_1.shape[0], rows, columns))
    optimized_valid_1_out = np.empty((y_valid_1.shape[0], rows, columns))
248 if do_hyparmod:
      cost_valid_1 = []
250   cost_valid_1_temp = 0
      cost_L2_valid_1 = []
252   cost_L2_valid_1_temp = 0
    for j in range(0,X_valid_1.shape[0]):
254   optimized_valid_1 = Net1.predict(X_valid_1[j,:,:])
      X_valid_1_out[j,:,:], y_valid_1_out[j,:,:], optimized_valid_1_out[j,:,:] =
                                          makefeatures_valid_1[j].
                                          rescale_and_reshape(X_valid_1[j,:,0],
                                          y_valid_1[j,:], optimized_valid_1, rows,
                                          columns)
256   if do_hyparmod:
        cost_valid_1_temp += Net1.cost_function(optimized_valid_1, y_valid_1[j,:])
258     cost_L2_valid_1_temp += Net1.cost_L2(optimized_valid_1, y_valid_1[j,:])
    if do_hyparmod:
260   cost_valid_1.append(cost_valid_1_temp/X_valid_1.shape[0])
      cost_L2_valid_1.append(cost_L2_valid_1_temp/X_valid_1.shape[0])
262
    ####### WRITE LOGFILE #######
264 if not os.path.exists('data/dataset' + str(dataset) + 'it' + str(n_cycles) + '/'):
      os.mkdir('data/dataset' + str(dataset) + 'it' + str(n_cycles) + '/', 0755)
266 file = open('data/dataset' + str(dataset) + 'it' + str(n_cycles) + '/parameters.log'
                                          , 'w')
    file.write('logfile\n\n')
268 file.write('base_inputdir : ' + base_inputdir + '\n')
    file.write('numinputdirs =  ' + str(numinputdirs) + '\n')
270 file.write('num_trainsets = ' + str(num_trainsets) + '\n')
    file.write('num_testsets =  ' + str(num_testsets) + '\n')
272 file.write('valid_inputdir_1 :  ' + valid_inputdir_1 + '\n')
    file.write('num_valid_1 = ' + str(num_valid_1) + '\n')
274

    file.write('n_output_features = ' + str(n_output_features) + '\n')
```

```
file.write('n_cycles =  ' + str(n_cycles) + '\n')

file.write('\n parameters' + '\n')
file.write('default_hidden_units =  ' + str(parameters[0][hidden_unit_index]) + '\n'
                                        )
file.write('default_eta = ' + str(parameters[1][eta_index]) + '\n')
file.write('default_batchsize = ' + str(parameters[2][batchsize_index]) + '\n')
file.write('modulated parameter : ' + parameter_labels[modpar] + '\n')
file.write(parameter_labels[modpar] + ' = ' + str(parameters[modpar]) + '\n')

file.write('\n cost in validation data' + '\n')
if do_hyparmod:
  file.write('cost_valid_1 =  ' + str(cost_valid_1) + '\n')
  file.write('cost_L2_valid_1 = ' + str(cost_L2_valid_1) + '\n')
file.close()

######## PLOT TRAIN ########
for j in range(0,X_train.shape[0]):
  desired_image_train = (X_train_out[j,:,:]-y_train_out[j,:,:])
  desired_image_train[desired_image_train<0] = 0
  predicted_image_train = (X_train_out[j,:,:]-optimized_train_out[j,:,:])
  predicted_image_train[predicted_image_train<0] = 0
  train_scale = np.max(X_train_out[j,:,:])
  y_diff_train = np.abs(y_train_out[j,:,:]-optimized_train_out[j,:,:])
  X_diff_train = np.abs(desired_image_train-predicted_image_train)
  Image.fromarray(min_custommax_scale(X_train_out[j,:,:], train_scale,
                                      selected_data_range)).save('data/dataset'
                                      + str(dataset) + 'it' + str(n_cycles) + '/
                                      train_' + str(j) + 'set' + str(dataset) +
                                      'it' + str(n_cycles) + '_original' + '.tif
                                      ')
  Image.fromarray(min_custommax_scale(y_train_out[j,:,:], train_scale,
                                      selected_data_range)).save('data/dataset'
                                      + str(dataset) + 'it' + str(n_cycles) + '/
                                      train_' + str(j) + 'set' + str(dataset) +
                                      'it' + str(n_cycles) + '_desired-scatter'
                                      + '.tif')
  Image.fromarray(min_custommax_scale(optimized_train_out[j,:,:], train_scale,
                                      selected_data_range)).save('data/dataset'
                                      + str(dataset) + 'it' + str(n_cycles) + '/
                                      train_' + str(j) + 'set' + str(dataset) +
                                      'it' + str(n_cycles) + '_predicted-scatter
                                      ' + '.tif')
  Image.fromarray(min_custommax_scale(desired_image_train, train_scale,
                                      selected_data_range)).save('data/dataset'
                                      + str(dataset) + 'it' + str(n_cycles) + '/
                                      train_' + str(j) + 'set' + str(dataset) +
                                      'it' + str(n_cycles) + '_desired-image' +
                                      '.tif')
  Image.fromarray(min_custommax_scale(predicted_image_train, train_scale,
                                      selected_data_range)).save('data/dataset'
                                      + str(dataset) + 'it' + str(n_cycles) + '/
                                      train_' + str(j) + 'set' + str(dataset) +
```

```
                                        'it' + str(n_cycles) + '_predicted-image'
                                        + '.tif')
  Image.fromarray(min_custommax_scale(y_diff_train, train_scale, selected_data_range
                                        )).save('data/dataset' + str(dataset) + '
                                        it' + str(n_cycles) + '/train_' + str(j) +
                                         'set' + str(dataset) + 'it' + str(
                                        n_cycles) + '_diff-scatter' + '.tif')
306 Image.fromarray(min_custommax_scale(X_diff_train, train_scale, selected_data_range
                                        )).save('data/dataset' + str(dataset) + '
                                        it' + str(n_cycles) + '/train_' + str(j) +
                                         'set' + str(dataset) + 'it' + str(
                                        n_cycles) + '_diff-image' + '.tif')
  train_enhance = np.max(y_train_out[j,:,:])
308 Image.fromarray(min_custommax_scale(y_train_out[j,:,:], train_enhance,
                                        selected_data_range)).save('data/dataset'
                                        + str(dataset) + 'it' + str(n_cycles) + '/
                                        train_' + str(j) + 'set' + str(dataset) +
                                        'it' + str(n_cycles) + '_desired-scatter'
                                        + '_enhanced' + '.tif')
  Image.fromarray(min_custommax_scale(optimized_train_out[j,:,:], train_enhance,
                                        selected_data_range)).save('data/dataset'
                                        + str(dataset) + 'it' + str(n_cycles) + '/
                                        train_' + str(j) + 'set' + str(dataset) +
                                        'it' + str(n_cycles) + '_predicted-scatter
                                        ' + '_enhanced' + '.tif')
310 Image.fromarray(min_custommax_scale(y_diff_train, train_enhance,
                                        selected_data_range)).save('data/dataset'
                                        + str(dataset) + 'it' + str(n_cycles) + '/
                                        train_' + str(j) + 'set' + str(dataset) +
                                        'it' + str(n_cycles) + '_diff-scatter' + '
                                        _enhanced' + '.tif')
  Image.fromarray(min_custommax_scale(X_diff_train, train_enhance,
                                        selected_data_range)).save('data/dataset'
                                        + str(dataset) + 'it' + str(n_cycles) + '/
                                        train_' + str(j) + 'set' + str(dataset) +
                                        'it' + str(n_cycles) + '_diff-image' + '
                                        _enhanced' + '.tif')
312
######## PLOT TEST ########
314 for j in range(0,X_test.shape[0]):
  desired_image_test = (X_test_out[j,:,:]-y_test_out[j,:,:])
316 desired_image_test[desired_image_test<0] = 0
  predicted_image_test = (X_test_out[j,:,:]-optimized_test_out[j,:,:])
318 predicted_image_test[predicted_image_test<0] = 0
  test_scale = np.max(X_test_out[j,:,:])
320 y_diff_test = np.abs(y_test_out[j,:,:]-optimized_test_out[j,:,:])
  X_diff_test = np.abs(desired_image_test-predicted_image_test)
322 Image.fromarray(min_custommax_scale(X_test_out[j,:,:], test_scale,
                                        selected_data_range)).save('data/dataset'
                                        + str(dataset) + 'it' + str(n_cycles) + '/
                                        test_' + str(j) + 'set' + str(dataset) + '
                                        it' + str(n_cycles) + '_original' + '.tif'
                                        )
```

```
    Image.fromarray(min_custommax_scale(y_test_out[j,:,:], test_scale,
                                        selected_data_range)).save('data/dataset'
                                        + str(dataset) + 'it' + str(n_cycles) + '/
                                        test_' + str(j) + 'set' + str(dataset) + '
                                        it' + str(n_cycles) + '_desired-scatter' +
                                         '.tif')
324 Image.fromarray(min_custommax_scale(optimized_test_out[j,:,:], test_scale,
                                        selected_data_range)).save('data/dataset'
                                        + str(dataset) + 'it' + str(n_cycles) + '/
                                        test_' + str(j) + 'set' + str(dataset) + '
                                        it' + str(n_cycles) + '_predicted-scatter'
                                         + '.tif')
    Image.fromarray(min_custommax_scale(desired_image_test, test_scale,
                                        selected_data_range)).save('data/dataset'
                                        + str(dataset) + 'it' + str(n_cycles) + '/
                                        test_' + str(j) + 'set' + str(dataset) + '
                                        it' + str(n_cycles) + '_desired-image' + '
                                        .tif')
326 Image.fromarray(min_custommax_scale(predicted_image_test, test_scale,
                                        selected_data_range)).save('data/dataset'
                                        + str(dataset) + 'it' + str(n_cycles) + '/
                                        test_' + str(j) + 'set' + str(dataset) + '
                                        it' + str(n_cycles) + '_predicted-image' +
                                         '.tif')
    Image.fromarray(min_custommax_scale(y_diff_test, test_scale, selected_data_range))
                                        .save('data/dataset' + str(dataset) + 'it'
                                         + str(n_cycles) + '/test_' + str(j) + '
                                        set' + str(dataset) + 'it' + str(n_cycles)
                                         + '_diff-scatter' + '.tif')
328 Image.fromarray(min_custommax_scale(X_diff_test, test_scale, selected_data_range))
                                        .save('data/dataset' + str(dataset) + 'it'
                                         + str(n_cycles) + '/test_' + str(j) + '
                                        set' + str(dataset) + 'it' + str(n_cycles)
                                         + '_diff-image' + '.tif')

330 Image.fromarray(min_max_scale(y_test_out[j,:,:], selected_data_range)).save('data/
                                        dataset' + str(dataset) + 'it' + str(
                                        n_cycles) + '/test_' + str(j) + 'set' +
                                        str(dataset) + 'it' + str(n_cycles) + '
                                        _desired-scatter' + '_enhanced' + '.tif')
    Image.fromarray(min_max_scale(optimized_test_out[j,:,:], selected_data_range)).
                                        save('data/dataset' + str(dataset) + 'it'
                                        + str(n_cycles) + '/test_' + str(j) + 'set
                                        ' + str(dataset) + 'it' + str(n_cycles) +
                                        '_predicted-scatter' + '_enhanced' + '.tif
                                        ')
332 Image.fromarray(min_max_scale(y_diff_test, selected_data_range)).save('data/
                                        dataset' + str(dataset) + 'it' + str(
                                        n_cycles) + '/test_' + str(j) + 'set' +
                                        str(dataset) + 'it' + str(n_cycles) + '
                                        _diff-scatter' + '_enhanced' + '.tif')
    Image.fromarray(min_max_scale(X_diff_test, selected_data_range)).save('data/
                                        dataset' + str(dataset) + 'it' + str(
```

```
                                        n_cycles) + '/test_' + str(j) + 'set' +
                                        str(dataset) + 'it' + str(n_cycles) + '
                                        _diff-image' + '_enhanced' + '.tif')
334
######## PLOT VALID_1 ########
336 for j in range(0,X_valid_1.shape[0]):
      desired_image_valid_1 = (X_valid_1_out[j,:,:]-y_valid_1_out[j,:,:])
338   desired_image_valid_1[desired_image_valid_1<0] = 0
      predicted_image_valid_1 = (X_valid_1_out[j,:,:]-optimized_valid_1_out[j,:,:])
340   predicted_image_valid_1[predicted_image_valid_1<0] = 0
      valid_1_scale = np.max(X_valid_1_out[j,:,:])
342   y_diff_valid_1 = np.abs(y_valid_1_out[j,:,:]-optimized_valid_1_out[j,:,:])
      X_diff_valid_1 = np.abs(desired_image_valid_1-predicted_image_valid_1)
344   Image.fromarray(min_custommax_scale(X_valid_1_out[j,:,:], valid_1_scale,
                                        selected_data_range)).save('data/dataset'
                                        + str(dataset) + 'it' + str(n_cycles) + '/
                                        valid_1_' + str(j) + 'set' + str(dataset)
                                        + 'it' + str(n_cycles) + '_original' + '.
                                        tif')
      Image.fromarray(min_custommax_scale(y_valid_1_out[j,:,:], valid_1_scale,
                                        selected_data_range)).save('data/dataset'
                                        + str(dataset) + 'it' + str(n_cycles) + '/
                                        valid_1_' + str(j) + 'set' + str(dataset)
                                        + 'it' + str(n_cycles) + '_desired-scatter
                                        ' + '.tif')
346   Image.fromarray(min_custommax_scale(optimized_valid_1_out[j,:,:], valid_1_scale,
                                        selected_data_range)).save('data/dataset'
                                        + str(dataset) + 'it' + str(n_cycles) + '/
                                        valid_1_' + str(j) + 'set' + str(dataset)
                                        + 'it' + str(n_cycles) + '_predicted-
                                        scatter' + '.tif')
      Image.fromarray(min_custommax_scale(desired_image_valid_1, valid_1_scale,
                                        selected_data_range)).save('data/dataset'
                                        + str(dataset) + 'it' + str(n_cycles) + '/
                                        valid_1_' + str(j) + 'set' + str(dataset)
                                        + 'it' + str(n_cycles) + '_desired-image'
                                        + '.tif')
348   Image.fromarray(min_custommax_scale(predicted_image_valid_1, valid_1_scale,
                                        selected_data_range)).save('data/dataset'
                                        + str(dataset) + 'it' + str(n_cycles) + '/
                                        valid_1_' + str(j) + 'set' + str(dataset)
                                        + 'it' + str(n_cycles) + '_predicted-image
                                        ' + '.tif')
      Image.fromarray(min_custommax_scale(y_diff_valid_1, valid_1_scale,
                                        selected_data_range)).save('data/dataset'
                                        + str(dataset) + 'it' + str(n_cycles) + '/
                                        valid_1_' + str(j) + 'set' + str(dataset)
                                        + 'it' + str(n_cycles) + '_diff-scatter' +
                                         '.tif')
350   Image.fromarray(min_custommax_scale(X_diff_valid_1, valid_1_scale,
                                        selected_data_range)).save('data/dataset'
                                        + str(dataset) + 'it' + str(n_cycles) + '/
                                        valid_1_' + str(j) + 'set' + str(dataset)
```

```
                                        + 'it' + str(n_cycles) + '_diff-image' + '
                                        .tif')

352   Image.fromarray(min_max_scale(y_valid_1_out[j,:,:], selected_data_range)).save('
                                        data/dataset' + str(dataset) + 'it' + str(
                                        n_cycles) + '/valid_1_' + str(j) + 'set' +
                                         str(dataset) + 'it' + str(n_cycles) + '
                                        _desired-scatter' + '_enhanced' + '.tif')
      Image.fromarray(min_max_scale(optimized_valid_1_out[j,:,:], selected_data_range)).
                                        save('data/dataset' + str(dataset) + 'it'
                                        + str(n_cycles) + '/valid_1_' + str(j) + '
                                        set' + str(dataset) + 'it' + str(n_cycles)
                                         + '_predicted-scatter' + '_enhanced' + '.
                                        tif')
354   Image.fromarray(min_max_scale(y_diff_valid_1, selected_data_range)).save('data/
                                        dataset' + str(dataset) + 'it' + str(
                                        n_cycles) + '/valid_1_' + str(j) + 'set' +
                                         str(dataset) + 'it' + str(n_cycles) + '
                                        _diff-scatter' + '_enhanced' + '.tif')
      Image.fromarray(min_max_scale(X_diff_valid_1, selected_data_range)).save('data/
                                        dataset' + str(dataset) + 'it' + str(
                                        n_cycles) + '/valid_1_' + str(j) + 'set' +
                                         str(dataset) + 'it' + str(n_cycles) + '
                                        _diff-image' + '_enhanced' + '.tif')
356

358 if do_hyparmod:
      fig, ax1 = plt.subplots()
360   ax1.plot(parameters[modpar], cost_train, 'r.', label = 'cost_train')
      ax1.plot(parameters[modpar], cost_test, 'r*', label = 'cost_test')
362   ax1.set_xlabel(parameter_labels[modpar])
      ax1.set_ylabel('cost')
364   ax1.tick_params(axis='y', colors='r')
      ax1.legend(loc = 1)
366   ax1.set_ylim(np.min([cost_train, cost_test])*0.7, np.max([cost_train, cost_test])*
                                        1.3)
      ax2 =ax1.twinx()
368   ax2.plot(parameters[modpar], cost_L2_train, 'b.', label = 'cost_L2_train')
      ax2.plot(parameters[modpar], cost_L2_test, 'b*', label = 'cost_L2_test')
370   ax2.set_ylabel('cost_L2')
      ax2.tick_params(axis='y', colors='b')
372   ax2.legend(loc = 2)
      ax2.set_ylim(np.min([cost_L2_train, cost_L2_test])*0.7, np.max([cost_L2_train,
                                        cost_L2_test])*1.3)
374   plt.tight_layout()
      plt.savefig(('data/dataset' + str(dataset) + 'it' + str(n_cycles) + '/cost_' +
                                        parameter_labels[modpar] + '.tif'), dpi =
                                        100, figsize = (15,10))
376   plt.savefig(('data/dataset' + str(dataset) + 'it' + str(n_cycles) + '/cost_' +
                                        parameter_labels[modpar] + '.pdf'), dpi =
                                        100, figsize = (15,10))
      #plt.show()
378   plt.close()
```

```
380 do_plot_overviews = False
if do_plot_overviews:
382   ####### PLOT TRAIN #######
  for j in range(0,X_train.shape[0]):
384     plt.subplot(231)
    plt.imshow(X_train_out[j,:,:], cmap = 'gray')
386     plt.title('original')
    plt.subplot(232)
388     plt.title('desired')
    plt.imshow(y_train_out[j,:,:], cmap = 'gray')
390     plt.subplot(233)
    plt.title('filtered after ' + str(n_cycles) + 'iterations')
392     plt.imshow(optimized_train_out[j,:,:], cmap = 'gray')
    plt.subplot(234)
394     plt.imshow(X_train_out[j,:,:], cmap = 'gray')
    plt.title('original')
396     plt.subplot(235)
    plt.title('desired')
398     plt.imshow(desired_image_train, cmap = 'gray')
    plt.subplot(236)
400     plt.title('filtered after ' + str(n_cycles) + 'iterations')
    plt.imshow(predicted_image_train, cmap = 'gray')
402     plt.suptitle('Traindata')
    plt.tight_layout()
404     plt.savefig(('data/dataset' + str(dataset) + 'it' + str(n_cycles) + '/
                                        train_overview_' + str(j) + 'set' + str(
                                        dataset) + 'it' + str(n_cycles) + '.tif'),
                                         dpi = 100, figsize = (15,10))
    plt.savefig(('data/dataset' + str(dataset) + 'it' + str(n_cycles) + '/
                                        train_overview_' + str(j) + 'set' + str(
                                        dataset) + 'it' + str(n_cycles) + '.pdf'),
                                         dpi = 100, figsize = (15,10))
406   # plt.show()
    plt.close()
408   ####### PLOT TEST #######
  for j in range(0,X_test.shape[0]):
410     plt.subplot(231)
    plt.imshow(X_test_out[j,:,:], cmap = 'gray')
412     plt.title('original')
    plt.subplot(232)
414     plt.title('desired')
    plt.imshow(y_test_out[j,:,:], cmap = 'gray')
416     plt.subplot(233)
    plt.title('filtered after ' + str(n_cycles) + 'iterations')
418     plt.imshow(optimized_test_out[j,:,:], cmap = 'gray')
    plt.subplot(234)
420     plt.imshow(X_test_out[j,:,:], cmap = 'gray')
    plt.title('original')
422     plt.subplot(235)
    plt.title('desired')
424     plt.imshow(desired_image_test, cmap = 'gray')
    plt.subplot(236)
```

```
426     plt.title('filtered after ' + str(n_cycles) + 'iterations')
        plt.imshow(predicted_image_test, cmap = 'gray')
428     plt.suptitle('Testdata')
        plt.tight_layout()
430     plt.savefig(('data/dataset' + str(dataset) + 'it' + str(n_cycles) + '/
                                        test_overview_' + str(j) + 'set' + str(
                                        dataset) + 'it' + str(n_cycles) + '.tif'),
                                         dpi = 100, figsize = (15,10))
        plt.savefig(('data/dataset' + str(dataset) + 'it' + str(n_cycles) + '/
                                        test_overview_' + str(j) + 'set' + str(
                                        dataset) + 'it' + str(n_cycles) + '.pdf'),
                                         dpi = 100, figsize = (15,10))
432   # plt.show()
        plt.close()
434   ######## PLOT VALID_1 ########
      for j in range(0,X_valid_1.shape[0]):
436     plt.subplot(231)
        plt.imshow(X_valid_1_out[j,:,:], cmap = 'gray')
438     plt.title('original')
        plt.subplot(232)
440     plt.title('desired')
        plt.imshow(y_valid_1_out[j,:,:], cmap = 'gray')
442     plt.subplot(233)
        plt.title('filtered after ' + str(n_cycles) + 'iterations')
444     plt.imshow(optimized_valid_1_out[j,:,:], cmap = 'gray')
        plt.subplot(234)
446     plt.imshow(X_valid_1_out[j,:,:], cmap = 'gray')
        plt.title('original')
448     plt.subplot(235)
        plt.title('desired')
450     plt.imshow(desired_image_valid_1, cmap = 'gray')
        plt.subplot(236)
452     plt.title('filtered after ' + str(n_cycles) + 'iterations')
        plt.imshow(predicted_image_valid_1, cmap = 'gray')
454     plt.suptitle('Validdata')
        plt.tight_layout()
456     plt.savefig(('data/dataset' + str(dataset) + 'it' + str(n_cycles) + '/
                                        valid_1_overview_' + str(j) + 'set' + str(
                                        dataset) + 'it' + str(n_cycles) + '.tif'),
                                         dpi = 100, figsize = (15,10))
        plt.savefig(('data/dataset' + str(dataset) + 'it' + str(n_cycles) + '/
                                        valid_1_overview_' + str(j) + 'set' + str(
                                        dataset) + 'it' + str(n_cycles) + '.pdf'),
                                         dpi = 100, figsize = (15,10))
458   # plt.show()
        plt.close()
```

A.2.7 Die Klasse zur Merkmalerstellung

convolve_class.py

```
import numpy as np
from scipy import signal
from scipy.ndimage import generic_filter
from scipy.ndimage import filters
import matplotlib.pyplot as plt

class convolveWork(object):

  def __init__(self, X, y):
    """

    Parameters
    ----------
    X : array, shape = [rows, columns]
      input image
    y : array, shape = [rows, columns]
      desired output image
    """
    self.X = X
    self.y = y
    self.features = []
    self.X_rescale = 1
    self.y_rescale = 1

  def maxpool(self, kernelsizex = 3, kernelsizey = 3):
    """give pixel the maximum value of area kernelsizex*kernelsizey around it

    Parameters
    ----------
    X : array, shape = [rows, columns]
      input image
    kernelsizex : int
      rows the kernel is regarding
    kernelsizey : int
      columns the kernel is regarding

    Returns
    -------
    pooled : array, shape = [rows, columns]
      maximum pooled matrix
    """
    pooled = filters.maximum_filter(self.X, size = (kernelsizex, kernelsizey), mode
                                          = 'constant')
    return pooled

  def minpool(self, kernelsizex = 3, kernelsizey = 3):
    """pgive pixel the minimal value of area kernelsizex*kernelsizey around it

    Parameters
    ----------
```

```
    X : array, shape = [rows, columns]
      input image
    kernelsizex : int
      rows the kernel is regarding
    kernelsizey : int
      columns the kernel is regarding

    Returns
    -------
    pooled : array, shape = [rows, columns]
      minimum pooled matrix
    """
    pooled = filters.minimum_filter(self.X, size = (kernelsizex, kernelsizey), mode
                                          = 'constant')
    return pooled

  def meanpool(self, kernelsizex = 3, kernelsizey = 3):
    """give pixel the mean value of area kernelsizex*kernelsizey around it

    Parameters
    ----------
    X : array, shape = [rows, columns]
      input image
    kernelsizex : int
      rows the kernel is regarding
    kernelsizey : int
      columns the kernel is regarding

    Returns
    -------
    pooled : array, shape = [rows, columns]
      mean pooled matrix
    """
    pooled = filters.uniform_filter(self.X, size = (kernelsizex, kernelsizey), mode
                                          = 'constant')
    return pooled

  def stdpool(self, kernelsizex = 3, kernelsizey = 3):
    #SOURCE: https://nickc1.github.io/python,/matlab/2016/05/17/Standard-Deviation-(
                                          Filters)-in-Matlab-and-Python.html
    """give pixel the standard-deviation value of area kernelsizex*kernelsizey
                                          around it

    Parameters
    ----------
    X : array, shape = [rows, columns]
      input image
    kernelsizex : int
      rows the kernel is regarding
    kernelsizey : int
      columns the kernel is regarding

    Returns
```

```
98      -------
        pooled : array, shape = [rows, columns]
100       standard-deviation pooled matrix
        """
102     X = self.X + np.random.rand(self.X.shape[0], self.X.shape[1])*1e-5    #to not
                                            get nans in the np.sqrt() call
        meanX = filters.uniform_filter(X, size = (kernelsizex, kernelsizey), mode = '
                                            constant')   #calculate the mean
104     meanXsq = filters.uniform_filter(X*X, size = (kernelsizex, kernelsizey), mode =
                                            'constant')   #calculate the mean of the
                                            squares
        return np.sqrt(meanXsq-meanX*meanX)   #calculate the standard deviaton
106
    def conv_w_kernel(self, kernel):
108   """convolvle the image X with the kernel

110     Parameters
        ----------
112     X : array, shape = [rows, columns]
          input image
114     kernel: array, shape = [kernelsizex, kernelsizey]
          the kernel to convolve with
116
        Returns
118     -------
        convolved : array, shape = [rows, columns]
120       convolvedd image
        """
122     convolved = signal.convolve2d(self.X, kernel, mode = 'same')
        return convolved
124
    def vectorize(self, do_scale = True, set_scale = False, custom_scale = False,
                                            X_cuscale = 1, y_cuscale = 1):
126   """vectorize image and features for better handling

128     Parameters
        ----------
130     do_scale : Bool
          scale the pixel values between 0 and 1 or not
132
        Returns
134     -------
        X_vectorized : array, shape = [n_pixels, n_features+1]
136       with n_pixels = rows*columns
        y_vectorized : array, shape = [n_pixels]
138     """
        self.X_vectorized = np.empty((self.X.shape[0]*self.X.shape[1], len(self.features
                                            )+1))
140     self.X_vectorized[:,0] = self.X.reshape((self.X.shape[0]*self.X.shape[1]))
        i = 1
142     for feature in self.features:
          self.X_vectorized[:,i] = feature.reshape(((self.X.shape[0]*self.X.shape[1])))
                                              #will raise error, if feature dimension
```

```
                                        mismatches picture dimension
      i+=1
    self.y_vectorized = self.y.reshape(self.y.shape[0]*self.y.shape[1])
    if do_scale:
      if set_scale:
        self.X_rescale = np.max(self.X_vectorized, axis = 0)
        self.y_rescale = np.max(self.y_vectorized)
      if custom_scale:
        self.X_rescale = X_cuscale
        self.y_rescale = y_cuscale
      self._scale()
    return self.X_vectorized, self.y_vectorized, self.X_rescale, self.y_rescale

  def _scale(self):
    """scale the data to the interval [0,1]
    """
    self.y_vectorized *= 1/self.y_rescale
    self.X_vectorized *= 1/self.X_rescale
    return self

  def rescale_and_reshape(self, X, y, optimized, rows, columns):
    """ rescale the data to the original absolute bandwidth

    Parameters
    ----------
    X : array, shape = [n_pixels]
      the input image, the first dimension of the feature matrix
    y : array, shape = [n_pixels]
      the desired image
    optimized : array, shaoe = [n_pixels]
      the output image of the NetWork
    max_scale : Bool
      scale each image to its own maximum
    data_range : str
      Which datatype is selected for the max-scaling

    Returns
    -------
    X_out : array, shape = [rows, columns]
      the input image, the first dimension of the feature matrix
    y_out : array, shape = [rows, columns]
      the desired image
    optimized_out : array, shaoe = [rows, columns]
      the output image of the NetWork

    """
    X_out = (self.X_rescale[0]*X).reshape((rows, columns))
    y_out = (self.y_rescale*y).reshape((rows, columns))
    optimized[optimized < 0] = 0
    optimized_out = (self.y_rescale*optimized).reshape((rows, columns))
    return X_out, y_out, optimized_out
```

A.2.8 Die Klasse des neuronalen Netzes

neuralnet_class.py

```
import numpy as np
from scipy import signal
from matplotlib import pyplot as plt

class NetWork(object):
  """

  Parameters
  ----------
  hidden_units : array, shape = [n_layers]
    array of the number of units in each layer, at index 0 it is equal to n_features
  L2 : float (default: 0.)
    the factor for L2-weight-regularization
  eta : float (default: 0.001)
    the learning rate
  cylces : int (default 10)
    the number of cycles to train the weights
  batchsize : int (default 1)
    the number of samples to consider before adapting the weights
  act_func : str (default:'logistic')
    selects the activation function
  seed : give a special random seed to make outputs reproducable
  """
  def __init__(self, hidden_units = [], L2 = 0., eta = 0.01, cycles = 1000,
                                        batchsize = 1, shuffle = False, act_func =
                                         'logistic', seed = None):
    self.hidden_units = hidden_units
    self.L2 = L2
    self.eta = eta
    self.cycles = cycles
    self.batchsize = batchsize
    self.shuffle = shuffle
    self.act_func = act_func
    self.random = np.random.RandomState(seed)

  def _activation_function(self, Z):
    """return the selected activation function

    Parameters
    ----------
    Z : array, shape = [n_units]
      Net input of the previous layer

    Returns
    -------
    phi : array, shape = [n_units]
      activation function of the net input
    """
    if self.act_func == 'logistic':
      """Compute logistic function"""
```

```
48        phi = 1. / (1. + np.exp(-Z))
          return phi
50      elif self.act_func == 'softmax':
          """Compute softmax function"""
52        phi = np.exp(Z)/np.sum(np.exp(Z))
          return phi
54      elif self.act_func == 'tanh':
          """Compute tangens hyperbolicus"""
56        phi = np.tanh(Z)
          return phi
58      else:
          print('ERROR: invalid activation function')
60
    def _initialize_weights(self):
62    """give the weights-matrices the layer-dependant shape and initialze with small
                                          random(gauss) values

64    Parameters
      ----------
66
      Returns
68    -------
      self.W : list, shape = [n_layers]
70      Gewichte, auf kleine Zufallszahlen initialisiert mit
        self.W[i] : array, shape[hidden_units[i], hidden_units[i+1]]
72    self.b : list, shape = [n_layers]
        Offset, auf null initialisiert, mit
74      self.b[i] : array, shape[hidden_units[i]]
      """
76    self.W = []
      self.b = []
78    for i in range(0, self.hidden_units.shape[0]-1):
        W_i = np.random.normal(loc=0.0, scale=0.1, size=(self.hidden_units[i], self.
                                          hidden_units[i+1]))   #weight matrices
                                          initialized with small random numbers
80      b_i = np.zeros(self.hidden_units[i+1])    #bias always initialized with zeros
        self.W.append(W_i)
82      self.b.append(b_i)
      return self
84
    def _forward_propagation(self, X):
86    """calculate the NetWorks's ouput by forward propagating the input values
                                          through each layer

88    Parameters
      ----------
90    X : array, shape = [n_samples, n_features+1]
        Array of input-values
92
      Returns
94    -------
      A : list, shape = [n_layers]
96      resulting values of the forward propagation with
```

```
      A[i] : array, shape = [n_samples, hidden_units[i]]
98    """
      A = []    #initialize A as a list
100   A.append(X)   #copy input values X as the first layer values
      for i in range(0, self.hidden_units.shape[0]-1):
102     Z = np.dot(A[-1], self.W[i]) + self.b[i]    #calculate the net-input for layer
                                                    i+1 including the bias
        A.append(self._activation_function(Z))    #calculate the activation
104   return A

106 def cost_function(self, y, a_output):
      """calculate the value of the cost function, to the current output of the
                                                NetWork
108
      Parameters
110   ----------
      y : array, shape = [n_samples]
112     the desired output values
      a_output: array, shape = [n_samples]
114     the calculated output values

116   Retturns
      --------
118   cost : float
        the cost of the curent values
120   """
      #L2_reg = self.L2 * np.sum(self.W**2)   #calculate the L2-regularizating part
122   #L2_reg = self.L2*sum([k**2 for k in self.W])
      #cost = np.sum(-y*np.log(a_output)-(1-y)*np.log(1-a_output)) + L2_reg
124   cost = np.sum(np.abs(y-a_output))/y.shape[0]
      return cost
126
    def cost_L2(self, y, a_output):
128   """calculate the value of the L2 regularization cost, to the current weights of
                                                the NetWork

130   Parameters
      ----------
132   y : array, shape = [n_samples]
        the desired output values
134   a_output: array, shape = [n_samples]
        the calculated output values
136
      Retturns
138   --------
      L2_reg : float
140     the cost of the curent weights
      """
142   L2_reg = 0
      for i in range(0,self.hidden_units.shape[0]-1):
144     L2_reg += np.sum(self.W[i]**2)
      #print('len(self.W)', len(self.W))
146   #print('len([k**2 for k in self.W])', len([k**2 for k in self.W]))
```

```
    #L2_reg = np.sum([k**2 for k in self.W])#*self.L2   #calculate the L2-
                                                regularizating part
148   return L2_reg

150 def predict(self, X):
    """predict output values
152
    Parameters
154 ----------
    X : array, shape = [n_samples, n_features+1]
156   Array of input-values

158 Returns
    -------KORRR
160 A_output : array, shape = [n_samples, hidden_units[:]]
      resulting values of the forward propagation, rounded to integer values
162 """
    A = self._forward_propagation(X)    #get calculated output
164 #A_output = np.around(A[-1])  #A[-1].astype(int)  #determine predicted  integer
                                                values
    #check this!!!!! Raschka is different! Why ? Why am i doing this= axis = 1 ??
166 return A[-1] #_output

168 def _backward_propagation(self, A, y):
    """backpropagate the error of the forward propagation to adapt the weights
170
    Parameters
172 ----------
    A : list, shape = [n_layers]
174   resulting values of the forward propagation with
      A[i] : array, shape = [n_samples, hidden_units[i]]
176 y : array, shape = [n_samples]
      desired values
178
    Returns
180 -------
    self.W : list, shape = [n_layers]
182   updated set of weights
      self.W[i] : array, shape[hidden_units[i-1], hidden_units[i]]
184 """
    delta = (A[-1].T-y)*A[-1].T*(1-A[-1].T)   #calculate delta for last layer
186 dJdW = np.dot(A[-2].T, delta.T)   #calculate dJdW for last layer
    self.W[-1] -= self.eta * dJdW   #adapt weights of the last layer
188 for i in range(1,len(A)-1):
      delta = np.sum(np.dot(self.W[-i], delta), axis = 0)*A[-(i+1)].T*(1-A[-(i+1)].T
                                                )   #iterative calculation of delta for
                                                layer i
190   dJdW = np.dot(A[-(i+2)].T, delta.T)   #calculate dJdW for layer i
      self.W[-(i+1)] -= self.eta * dJdW   #adapt weights of layer i
192   return self


194
  def fit(self, X, y):
```

```
196     """pass data through network and adapt weights

198     Parameters
        ----------
200     X : array, shape = [n_pictures, n_pixels, n_features+1]
          training data
202     y : array, shape = [n_pictures, n_pixels]
        """
204     self._initialize_weights()    #first, initialize the weights

206     A_out = np.empty(y.shape)   #initialize the ouput matrix
        for i in range(self.cycles):    #loop over the training cycles
208       print('cycle:', i)
          for picture_index in range(0, X.shape[0]):    #loop over all pictures
210         pixel_index = np.arange(X.shape[1])   #get the indexes of the pixels to
            if self.shuffle:
212           np.random.shuffle(pixel_index)    #shuffle them
            current_index = 0
214         while current_index < (X.shape[1]-self.batchsize):    #go through it in
                                              batchsized steps
              batch = pixel_index[current_index:current_index + self.batchsize]
216           current_index += self.batchsize
              A = self._forward_propagation(X[picture_index, batch, :])   #forward the
                                              batch
218           self._backward_propagation(A, y[picture_index, batch])    #adapt the
                                              weights
            A_out[picture_index, :, None] = self.predict(X[picture_index,:,:])    #save
                                              the predicted image
220     return A_out
```

A.3 GATE-Skripte

A.3.1 Beispiel eines Skriptes

thirdCT_newspectra_thesis.mac

```
####### EXECUTE & SOURCE EXTERNAL FILES #######
/control/execute thirdalias_cluster.mac
/gate/geometry/setMaterialDatabase GateMaterials.db

####### SET THE VISUALIZATION #######
/vis/disable
#/control/execute vis2.mac

####### DEFINE THE WORLD #######        # The world is the space in wich particles
    are tracked
/gate/world/geometry/setXLength {worldsize}     #size definition
/gate/world/geometry/setYLength {worldsize}
/gate/world/geometry/setZLength {worldsize}
/gate/world/setMaterial Air     # Material definition
/gate/world/vis/forceWireframe      #visualization parameters
/gate/world/vis/setColor black

```

```
####### DEFINE THE DETECTOR #######
/gate/world/daughters/name CTscanner
/gate/world/daughters/insert box        #shape of the detector
/gate/CTscanner/placement/setTranslation {CTscannerPosX} {CTscannerPosY} {
    CTscannerPosZ} {CTscannerPosSCALE}        #position of the detector
/gate/CTscanner/geometry/setXLength {CTscannerSizeX}
/gate/CTscanner/geometry/setYLength {CTscannerSizeY}
/gate/CTscanner/geometry/setZLength {CTscannerSizeZ}
/gate/CTscanner/setMaterial {CTscannerMaterial}
/gate/CTscanner/vis/forceWireframe
/gate/CTscanner/vis/setColor white

####### DEFINE THE MODULES INSIDE THE DETECTOR #######
/gate/CTscanner/daughters/name module       #the module is a daughter volume of the
    detectror
/gate/CTscanner/daughters/insert box
/gate/module/geometry/setXLength {moduleSizeX}
/gate/module/geometry/setYLength {moduleSizeY}
/gate/module/geometry/setZLength {moduleSizeZ}
/gate/module/setMaterial {moduleMaterial}
/gate/module/vis/forceWireframe
/gate/module/vis/setColor white

####### DEFINE THE CLUSTERS INSIDE THE MODULES #######
/gate/module/daughters/name cluster
/gate/module/daughters/insert box
/gate/cluster/geometry/setXLength {clusterSizeX}
/gate/cluster/geometry/setYLength {clusterSizeY}
/gate/cluster/geometry/setZLength {clusterSizeZ}
/gate/cluster/setMaterial {clusterMaterial}
/gate/cluster/vis/forceWireframe
/gate/cluster/vis/setColor white

####### DEFINE THE PIXELS INSIDE THE CLUSTERS #######
/gate/cluster/daughters/name pixel
/gate/cluster/daughters/insert box
/gate/pixel/geometry/setXLength {pixelSizeX}
/gate/pixel/geometry/setYLength {pixelSizeY}
/gate/pixel/geometry/setZLength {pixelSizeZ}
/gate/pixel/setMaterial {pixelMaterial}
/gate/pixel/vis/forceSolid
/gate/pixel/vis/setColor red

# REPEAT THE PIXElS #
/gate/pixel/repeaters/insert cubicArray
/gate/pixel/cubicArray/setRepeatNumberX {pixelnumberX}      # the number of pixels
    in each dimension
/gate/pixel/cubicArray/setRepeatNumberY {pixelnumberY}
/gate/pixel/cubicArray/setRepeatNumberZ {pixelnumberZ}
/gate/pixel/cubicArray/setRepeatVector {pixelVector}        #the vector to repeat
    the pixels
/gate/pixel/cubicArray/autoCenter true
```

```
66 ######## ATTACH SYSTEM #######
   /gate/systems/CTscanner/module/attach module
68 /gate/systems/CTscanner/cluster_0/attach cluster
   /gate/systems/CTscanner/pixel_0/attach pixel
70
   ######## ATTACH THE DETECTING OBJECTS ########
72 /gate/pixel/attachCrystalSD


74


76
   ######## DEFINE THE PHANTOM ########
78 /gate/world/daughters/name waterCylinder        #name of the first part of the
       phantom
   /gate/world/daughters/insert cylinder       #shape
80 /gate/waterCylinder/placement/setRotationAxis 1 0 0     #define the placement
   /gate/waterCylinder/placement/setRotationAngle 90. deg
82 /gate/waterCylinder/geometry/setRmin 0. mm      #geometry definition
   /gate/waterCylinder/geometry/setRmax {CylinderRadius}
84 /gate/waterCylinder/geometry/setHeight {CylinderHeight}
   /gate/waterCylinder/setMaterial {CylinderMaterial}
86 /gate/waterCylinder/vis/forceWireframe
   /gate/waterCylinder/vis/setColor cyan
88
   /gate/waterCylinder/moves/insert rotation       #rotate the WaterCylinder and all
       its daughter volumes
90 /gate/waterCylinder/rotation/setSpeed {WaterCylinderRotation}       #the degrees of
       rotation per slice
   /gate/waterCylinder/rotation/setAxis 0 0 1
92
   /gate/waterCylinder/daughters/name Ball_mid
94 /gate/waterCylinder/daughters/insert sphere
   /gate/Ball_mid/placement/setTranslation {Ball_mid_Pos}
96 /gate/Ball_mid/geometry/setRmin 0. mm
   /gate/Ball_mid/geometry/setRmax {Ball_mid_Radius}
98 /gate/Ball_mid/setMaterial {Ball_mid_Material}
   /gate/Ball_mid/vis/forceSolid
100 /gate/Ball_mid/vis/setColor white

102 /gate/waterCylinder/daughters/name Ball_1
    /gate/waterCylinder/daughters/insert sphere
104 /gate/Ball_1/placement/setTranslation {Ball_1_Pos}
    /gate/Ball_1/geometry/setRmin 0. mm
106 /gate/Ball_1/geometry/setRmax {Ball_1_Radius}
    /gate/Ball_1/setMaterial {Ball_1_Material}
108 /gate/Ball_1/vis/forceSolid
    /gate/Ball_1/vis/setColor yellow
110
    /gate/waterCylinder/daughters/name Ball_2
112 /gate/waterCylinder/daughters/insert sphere
    /gate/Ball_2/placement/setTranslation {Ball_2_Pos}
114 /gate/Ball_2/geometry/setRmin 0. mm
    /gate/Ball_2/geometry/setRmax {Ball_2_Radius}
```

```
/gate/Ball_2/setMaterial {Ball_2_Material}
/gate/Ball_2/vis/forceSolid
/gate/Ball_2/vis/setColor yellow

/gate/waterCylinder/daughters/name Ball_3
/gate/waterCylinder/daughters/insert sphere
/gate/Ball_3/placement/setTranslation {Ball_3_Pos}
/gate/Ball_3/geometry/setRmin 0. mm
/gate/Ball_3/geometry/setRmax {Ball_3_Radius}
/gate/Ball_3/setMaterial {Ball_3_Material}
/gate/Ball_3/vis/forceSolid
/gate/Ball_3/vis/setColor yellow

/gate/waterCylinder/daughters/name Ball_4
/gate/waterCylinder/daughters/insert sphere
/gate/Ball_4/placement/setTranslation {Ball_4_Pos}
/gate/Ball_4/geometry/setRmin 0. mm
/gate/Ball_4/geometry/setRmax {Ball_4_Radius}
/gate/Ball_4/setMaterial {Ball_4_Material}
/gate/Ball_4/vis/forceSolid
/gate/Ball_4/vis/setColor yellow

/gate/waterCylinder/daughters/name Ball_5
/gate/waterCylinder/daughters/insert sphere
/gate/Ball_5/placement/setTranslation {Ball_5_Pos}
/gate/Ball_5/geometry/setRmin 0. mm
/gate/Ball_5/geometry/setRmax {Ball_5_Radius}
/gate/Ball_5/setMaterial {Ball_5_Material}
/gate/Ball_5/vis/forceSolid
/gate/Ball_5/vis/setColor yellow

/gate/waterCylinder/daughters/name Ball_6
/gate/waterCylinder/daughters/insert sphere
/gate/Ball_6/placement/setTranslation {Ball_6_Pos}
/gate/Ball_6/geometry/setRmin 0. mm
/gate/Ball_6/geometry/setRmax {Ball_6_Radius}
/gate/Ball_6/setMaterial {Ball_6_Material}
/gate/Ball_6/vis/forceSolid
/gate/Ball_6/vis/setColor yellow

# REPEAT PARTS OF THE PHANTOM #
/gate/Ball_1/repeaters/insert linear        # repeat all the balls
/gate/Ball_1/linear/setRepeatVector {BallsVector}
/gate/Ball_2/repeaters/insert linear
/gate/Ball_2/linear/setRepeatNumber {NumberofBalls}
/gate/Ball_2/linear/setRepeatVector {BallsVector}
/gate/Ball_3/repeaters/insert linear
/gate/Ball_3/linear/setRepeatNumber {NumberofBalls}
/gate/Ball_3/linear/setRepeatVector {BallsVector}
/gate/Ball_4/repeaters/insert linear
/gate/Ball_4/linear/setRepeatNumber {NumberofBalls}
/gate/Ball_4/linear/setRepeatVector {BallsVector}
/gate/Ball_5/repeaters/insert linear
```

```
/gate/Ball_5/linear/setRepeatNumber {NumberofBalls}
/gate/Ball_5/linear/setRepeatVector {BallsVector}
/gate/Ball_6/repeaters/insert linear
/gate/Ball_6/linear/setRepeatNumber {NumberofBalls}
/gate/Ball_6/linear/setRepeatVector {BallsVector}

/gate/world/daughters/name PhiBox
/gate/world/daughters/insert box
/gate/PhiBox/placement/setTranslation {PhiBoxPosX} {PhiBoxPosY} {PhiBoxPosZ} {
    PhiBoxPosSCALE}
/gate/PhiBox/geometry/setXLength {PhiBoxSizeX}
/gate/PhiBox/geometry/setYLength {PhiBoxSizeY}
/gate/PhiBox/geometry/setZLength {PhiBoxSizeZ}
/gate/PhiBox/setMaterial {PhiBoxMaterial}
/gate/PhiBox/vis/forceWireframe
/gate/PhiBox/vis/setColor cyan

/gate/PhiBox/daughters/name Phi
/gate/PhiBox/daughters/insert tessellated       # include a stl file as phantom
/gate/Phi/placement/setTranslation {PhiPos}
/gate/Phi/geometry/setPathToSTLFile {PhiPath}
/gate/Phi/setMaterial {PhiMaterial}

/gate/Phi/repeaters/insert cubicArray       # repeat the stl
/gate/Phi/cubicArray/setRepeatNumberX {PhinumberX}
/gate/Phi/cubicArray/setRepeatNumberY {PhinumberY}
/gate/Phi/cubicArray/setRepeatNumberZ {PhinumberZ}
/gate/Phi/cubicArray/setRepeatVector {PhiVector}
/gate/Phi/cubicArray/autoCenter true

/gate/PhiBox/moves/insert orbiting      # rotate the PhiBox
/gate/PhiBox/orbiting/setSpeed {PhiBoxRotation}
/gate/PhiBox/orbiting/setPoint1 0 1 0 cm
/gate/PhiBox/orbiting/setPoint2 0 0 0 cm

######## ATTACH THE PHANTOM ########
/gate/waterCylinder/attachPhantomSD
/gate/Ball_mid/attachPhantomSD
/gate/Ball_1/attachPhantomSD
/gate/Ball_2/attachPhantomSD
/gate/Ball_3/attachPhantomSD
/gate/Ball_4/attachPhantomSD
/gate/Ball_5/attachPhantomSD
/gate/Ball_6/attachPhantomSD
/gate/PhiBox/attachPhantomSD
/gate/Phi/attachPhantomSD

/vis/viewer/set/viewpointThetaPhi 130 60
/vis/viewer/panTo 0 0 mm

######## SET THE PHYSICS ########
/gate/physics/addPhysicsList emstandard_opt3
/gate/physics/Gamma/SetCutInRegion      world 1. cm
```

```
/gate/physics/Electron/SetCutInRegion   CTscanner 1. cm
222
######## SIMULATION STATISTICS ########
224 /gate/actor/addActor   SimulationStatisticActor stat
/gate/actor/stat/save  {savepath}/stat-ct.txt
226
######## INITIALISATION ########
228 /gate/run/initialize

230 ######## DIGITIZER ########
/gate/digitizer/Singles/insert adder
232 /gate/digitizer/Singles/insert readout
/gate/digitizer/Singles/readout/setDepth 2          #good?
234
######## SOURCE ########
236 /gate/source/addSource xraygun      #add the source
/gate/source/verbose 0
238 /gate/source/xraygun/gps/verbose 0
/gate/source/xraygun/gps/particle gamma     # define the particle
240 /gate/source/xraygun/gps/type Plane     # define the form
/gate/source/xraygun/gps/shape Rectangle        # define the shape
242 /gate/source/xraygun/gps/halfx {focusHalfSizeX}     # define the size
/gate/source/xraygun/gps/halfy {focusHalfSizeY}
244 /gate/source/xraygun/gps/centre {focusPosX} {focusPosY} {focusPosZ} {focusPosSCALE}
       # define the position
/gate/source/xraygun/gps/maxtheta {focusAngle}      # define the emitting angle
246 /gate/source/xraygun/gps/angtype iso
/gate/source/xraygun/gps/energytype Arb     # define the sources spectrum by
   profiding a histogram
248 /gate/source/xraygun/gps/histname arb
/gate/source/xraygun/gps/emin 3.00 keV
250 /gate/source/xraygun/gps/emax 600.00 keV
/gate/source/xraygun/gps/histpoint  .003 .386377193910233
252 /gate/source/xraygun/gps/histpoint  .0041639278557114  0.617314450096939
/gate/source/xraygun/gps/histpoint  .0053278557114229  0.893803494023178
254 /gate/source/xraygun/gps/histpoint  .0065917835671343  1.22408032839862
/gate/source/xraygun/gps/histpoint  .0077557114228457  1.60781684261882
```

Es wird der Großteil des Spektrums übersprungen.

```
1 /gate/source/xraygun/gps/histpoint  .594018036072144  0.028237121886558
/gate/source/xraygun/gps/histpoint  .595214428857716  0.022493538463034
3 /gate/source/xraygun/gps/histpoint  .596410821643287  0.016798061785091
/gate/source/xraygun/gps/histpoint  .597607214428858  0.011150449572943
5 /gate/source/xraygun/gps/histpoint  .598803607214429  0.005551163981842
/gate/source/xraygun/gps/histpoint  .600 0
7
/gate/source/xraygun/gps/arbint Lin
9 /gate/source/list

11
######## OUTPUT ########
13 # ROOT #
/gate/output/root/enable        # enable the root output
15 /gate/output/root/setFileName {savepath}/ct_results
```

```
/gate/output/root/setRootHitFlag 1
/gate/output/root/setRootSinglesFlag 1
/gate/output/root/setRootNtupleFlag 0
/gate/output/verbose 1

# Specific CT Image {savepath}
/gate/output/imageCT/enable     # enable the image output
/gate/output/imageCT/setFileName {savepath}/ct_projection
/gate/output/imageCT/verbose 0

######## DEFINE THE RANDOM ENGINE & SEED ########
# JamesRandom Ranlux64 MersenneTwister
/gate/random/setEngineName MersenneTwister
/gate/random/setEngineSeed auto

######## DEFINE THE NUMBER OF PARTICLES AND THE NUMBER OF SLICES TO PRODUCE ########
/gate/application/setTotalNumberOfPrimaries {NumberOfPrimaries}
/gate/application/setTimeSlice    {TimeSlice}
/gate/application/setTimeStart    {TimeStart}
/gate/application/setTimeStop     {TimeStop}
/gate/application/start
```

A.3.2 Beispiel einer Alias-Datei

thirdalias_thesis.mac

```
/control/alias worldsize  100. cm

/control/alias CTscannerPosX 0.
/control/alias CTscannerPosY 0.
/control/alias CTscannerPosZ 400.
/control/alias CTscannerPosSCALE mm
/control/alias CTscannerSizeX 150. mm
/control/alias CTscannerSizeY 200. mm
/control/alias CTscannerSizeZ .5 mm
/control/alias CTscannerMaterial Air

/control/alias moduleSizeX {CTscannerSizeX}
/control/alias moduleSizeY {CTscannerSizeY}
/control/alias moduleSizeZ {CTscannerSizeZ}
/control/alias moduleMaterial {CTscannerMaterial}

/control/alias clusterSizeX {CTscannerSizeX}
/control/alias clusterSizeY {CTscannerSizeY}
/control/alias clusterSizeZ {CTscannerSizeZ}
/control/alias clusterMaterial {CTscannerMaterial}

/control/alias pixelSizeX 1. mm
/control/alias pixelSizeY 1. mm
/control/alias pixelSizeZ 1. mm
/control/alias pixelMaterial Silicon
/control/alias pixelnumberX 150
/control/alias pixelnumberY 200
/control/alias pixelnumberZ 1
```

```
/control/alias pixelVector 1 1 .0 mm

/control/alias CylinderHeight 35. mm
/control/alias CylinderRadius 13. mm
/control/alias CylinderMaterial Water
/control/alias WaterCylinderRotation 24. deg/s

/control/alias Ball_mid_Pos   0. 0. 0. mm
/control/alias Ball_mid_Radius 5. mm
/control/alias Ball_mid_Material Aluminium
/control/alias Ball_1_Pos   9. 0.  0. mm
/control/alias Ball_1_Radius 3. mm
/control/alias Ball_1_Material Tungsten
/control/alias Ball_2_Pos  -9. 0.  0. mm
/control/alias Ball_2_Radius 3. mm
/control/alias Ball_2_Material Tungsten
/control/alias Ball_3_Pos   6. -6. 0. mm
/control/alias Ball_3_Radius 3. mm
/control/alias Ball_3_Material SpineBone
/control/alias Ball_4_Pos  -6. -6. 0. mm
/control/alias Ball_4_Radius 3. mm
/control/alias Ball_4_Material Germanium
/control/alias Ball_5_Pos  -6. 6.  0. mm
/control/alias Ball_5_Radius 3. mm
/control/alias Ball_5_Material SpineBone
/control/alias Ball_6_Pos   6. 6.  0. mm
/control/alias Ball_6_Radius 3. mm
/control/alias Ball_6_Material Germanium
/control/alias BallsVector 0. 0. 8. mm
/control/alias NumberofBalls 4

/control/alias PhiBoxPosX 0.
/control/alias PhiBoxPosY 0.
/control/alias PhiBoxPosZ 20.
/control/alias PhiBoxSizeX 50. mm
/control/alias PhiBoxSizeY 50. mm
/control/alias PhiBoxSizeZ 5. mm
/control/alias PhiBoxPosSCALE mm
/control/alias PhiBoxMaterial Water
/control/alias PhiBoxRotation 180. deg/s
/control/alias PhiPos 0. 0. 0. mm
/control/alias PhiVector 6. 6. 0. mm
/control/alias PhinumberX 6
/control/alias PhinumberY 6
/control/alias PhinumberZ 1
/control/alias PhiMaterial Tungsten
/control/alias PhiPath  phi_8mm.stl

/control/alias focusHalfSizeX .025 mm
/control/alias focusHalfSizeY .025 mm
/control/alias focusPosX 0.
/control/alias focusPosY 0.
/control/alias focusPosZ -100.
```

```
/control/alias focusPosSCALE mm
/control/alias focusAngle 15. deg

/control/alias TimeSlice 1.  s
/control/alias TimeStart 0.   s
/control/alias TimeStop 30.  s


/control/alias NumberOfPrimaries 100000000
/control/alias version 30slices_9
/control/alias basepath /media/Data/Data_sjung/simulations//thirdCT
/control/alias savepath {basepath}_{NumberOfPrimaries}_{version}
```

A.3.3 Datei zur Visualisierung

vis2.mac

```
#   V I S U A L I S A T I O N
/vis/open OGLSX
/vis/viewer/set/viewpointThetaPhi 60 60
/vis/viewer/zoom 1.5
/vis/viewer/zoom 5
/vis/drawVolume
/vis/viewer/flush
/tracking/verbose 0
/tracking/storeTrajectory 1
/vis/scene/add/trajectories
/vis/scene/endOfEventAction accumulate
```

Erklärung

gem. §14 Abs. 7 Prüfungsordnung vom 15.04.2013

Hiermit erkläre ich, dass ich die von mir eingereichte Abschlussarbeit (Bachelor-Thesis) selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Stellen der Abschlussarbeit, die anderen Werken dem Wortlaut oder Sinn nach entnommen wurden, in jedem Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Ort / Datum	Simon Jung

Erklärung

Hiermit erkläre ich mich damit einverstanden, dass meine Abschlussarbeit (Bachelor-Thesis) wissenschaftlich interessierten Personen oder Institutionen und im Rahmen von externen Qualitätssicherungsmaßnahmen des Studienganges zur Einsichtnahme zur Verfügung gestellt werden kann.
Korrektur- oder Bewertungshinweise in meiner Arbeit dürfen nicht zitiert werden.

Ort / Datum	Simon Jung