UNIVERSITÄT STUTTGART
INSTITUT FÜR TECHNISCHE OPTIK
PROFESSOR DR. S. REICHELT

Forschungsarbeit

Arithmetik für Bildinhalte mit Neuronalen Netzen

Mengze Lu

Studiengang Elektromobilität

Matrikelnummer 3110638

30. Juni 2023

Prüfer:

Prof. Dr. Alois Herkommer

Betreuer:

Alexander Birk, M. Sc.

Xiwei Wang, M. Sc.

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Forschungsarbeit selbstständig verfasst habe.

Ich versichere, dass ich keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe. Dies gilt auch für alle verwendeten Abbildungen und Formeln.

Ich versichere, dass das elektronische Exemplar mit den anderen abgegebenen Exemplaren übereinstimmt.

Stuttgart, den 30. Juni 2023

_____

Unterschrift

Inhaltsverzeichnis

Abkürzungsverzeichnis

KNN Künstliche neuronale Netze

NN neuronalen Netzen

AE Autoencoder

ReLU Rectified Linear Unit

MSE Mean Squared Error

MAE Mean Absolute Error

VAE Variational Autoencoder

CAE Convolutional Autoencoder

SAE Sparse Autoencoder

RAE ResNet Autoencoder

C-RAE Convolutional ResNet Autoencoder

CNN Convolutional Neural Networks

SSIM structural similarity index

1 Motivation und Zielsetzung

In den letzten Jahren sind Künstliche neuronale Netze (KNN) bzw. Deep-Learning in vielen Forschungsbereichen immer beliebter geworden. Sie sind in der large, komplexe Aufgaben zu lösen, bei denen traditionalen Methoden nicht oder begrenzt erfolgreich sein können. Im Vergleich zu traditionalen Methoden liegt die hauptvorteil des KNNs daran, dass ein neuronales Netz von Daten selbst lernen kann. KNN machen derzeit insbesondere in Bildverarbeitungsbereich rasante Fortschritte. KNN werden oft genutzt, die Merkmale den Bildern zu erkennen und zu generalisieren. Einige Verwendungen im Bereich Bildverarbeitung sind: In [21] wurde ein große Convolutional Neural Networks (CNN) trainiert, um die Bilder zu klassifizieren. In [29] wurde ein System basierend auf CNN für Objekte-Erkennung dargestellt. Ein Autoencoder wurde in [38] für die Bildentrauschung verwendet. Diese Erforschungen belegen den Erfolg und die Bedeutung des Einsatzes von neuronalen Netzen (NN).

Obwohl das KNN eigenständig Informationen extrahiert und eine Lösung erzeugt, ist das innere Verhalten eines neuronalen Netzes dem Benutzer nicht direkt zugänglich. Nur die Eingabedaten und die Ausgabedaten für den Nutzer verfügbar sind. Es ist schwer zu nachvollziehen, wie die Daten im NN transformiert werden. Das liegt an dem 'Black-Box'-Charakter des KNN. Die Black Box ist ein komplexes mathematisches Modell, zu dem man keinen Einblick hat und dessen Funktionsweise unbekannt ist. Die Daten in der Black Box können als latente Vektoren bezeichnet werden. Solche latente Vektoren sind höherdimensionalen Daten und ihre Bedeutung für den Menschen ist unverständlich. In neuronalen Netz kann die Froschern viele Schwierigkeit betroffen. Die Struktur des Netzes mit komplexen Schichten kann schwer nachvollziehbar sein. Es ist schwer die Aktivitäten und Gewichtungen eine Netzes in einer für Menschen verständlichen Form zu präsentieren. Diese Problemen wird als Black-Box-Problem benennt. Ein Verständnis zur latenten Vektoren kann hilfreich für Black-Box-Problem zu lösen. Durch das Verständnis der Funktionsweise der 'Black-Box' können gezielte Verbesserungen an der Netzarchitektur vorgenommen werden. Die Kontrolle über latente Vektoren bietet mehrere Vorteile. Es ermöglicht eine gezielte Steuerung spezifischer Merkmale eines Datensatzes. Beispielsweise können bestimmte Merkmale gezielt hinzugefügt werden, um Bilder zu generieren, die diese Merkmale enthalten. Durch die Steuerung von Merkmalen können Forscher besser verstehen, welche Dimensionen im latenten Raum für bestimmte Veränderungen in den Originaldaten verantwortlich sind.

Das Ziel dieser Arbeit besteht darin, die latente Vektoren im latenten Raum zu berechnen, um eine gewünschte Ausgabe zu erzielen. Hierfür wird ein spezielles neuronales

Netzwerk verwendet, insbesondere ein Convolutional ResNet Autoencoder (C-RAE). Dieser wird mit verschiedenen Datensätzen, die in dieser Arbeit generiert werden, trainiert, um die Addition oder Subtraktion im latenten Raum zu untersuchen. Der C-RAE soll in der Lage sein, mithilfe der berechneten komprimierten Eingabedaten ein neues Bild zu generieren, in dem alle Merkmale der Eingangsbilder auftauchen (bei der Addition) oder bestimmte Merkmale der Eingangsbilder isolieren (bei der Subtraktion).

Die vorliegende Arbeit ist wie folgt strukturiert: Kapitel 2 bietet eine Einführung in die grundlegenden Konzepte, die für das Verständnis des KNN und die Funktionsweise des C-RAE erforderlich sind. Die Implementierung des entwickelten C-RAE wird in Kapitel 3 detailliert beschrieben. Kapitel 4 präsentiert und analysiert die durchgeführten Experimente sowie die erzielten Ergebnisse. Abschließend werden in Kapitel 5 die wichtigsten Erkenntnisse zusammengefasst und ein Ausblick für zukünftige Forschungsbereiche gegeben.

2 Grundlagen

In diesem Kapitel werden die zugrundeliegenden Konzepte und Methoden behandelt. Es bietet eine Übersicht über die Grundlagen der neuronalen Netze und des ResNet-Autoencoders. in Abschnitt 2.1 würde Künstliche neuronale Netze und die Hyperparameter für das Training erklärt. die Grundlagen des Autoencoders werden in Abschnitt 2.2 erläutert. In Abschnitt 2.3 wird das ResNet Autoencoder im Detail beschrieben. Der SSIM-Wert und der Sobel-Filter werden in Abschnitt 2.4 erklärt.

2.1 Künstliche neuronale Netze

Künstliche neuronale Netze (KNN) sind rechnerische Verarbeitungssysteme, die zu einem Teilbereich des maschinellen Lernens gehören. Die Netze haben die Fähigkeit, wie ein biologisches Nervensystem zu lernen und komplizierte Problem zu lösen. Das Netzwerk lernt einen Zusammenhang zwischen Ein- und Ausgangsdaten durch das Training. Im Training werden seine internen Gewichte optimiert, um Muster in den Daten zu identifizieren und Vorhersagen zu treffen.

Ein künstliche Neuron ist das Grundbaustein der künstliche neuronale Netze. Das mathematische Modell eines künstlichen Neurons wird in Abbildung 1 dargestellt. Es besteht aus vier Elemente: die Gewichtung, die Übertragungsfunktion, die Aktivierungsfunktion und der Schwellwert. Die Ein- und Ausgangsdaten werden als x_i und o_j beschrieben. Die Gewichte der Verbindung von Neuron i zu Neuron j werden mit w_{ij} bezeichnet, die die Wichtigkeit den Eingangsdaten des Neurons für die Ausgangsdaten des Neurons beeinflussen. Die Netzeingabe des Neurons wird durch die Übertragungsfunktion anhand der Gewichtung der Eingaben berechnet. Eine beliebte Übertragungsfunktion ist die gewichtete Summe. Die Netzeingabe werden in der Literatur [20] wie folgt definiert:

$$net_j = \sum_{i=1}^{n} w_{ij} x_i \tag{2.1}$$

wobei n ist die Anzahl des Eingabe. Die Netzeingabe wird anschließend durch eine Aktivierungsfunktion φ transformiert und als Ausgabe dargestellt. Die Aktivierungsfunktion berechnet abhängig von Netzeingabe, wie stark ein Neuron aktiviert ist [20]. Die

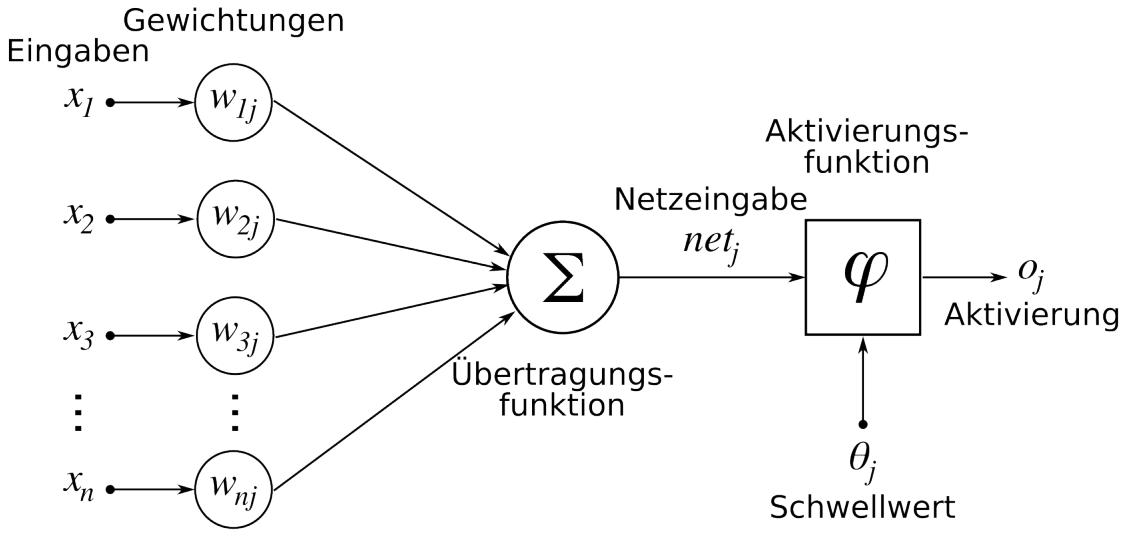


Abbildung 1: Das mathematische Modells eines künstlichen Neurons. [8]

Ausgabe des Neurons wird wie folgt definiert:

$$o_j = \varphi(\sum_{i=1}^{n} w_{ij} x_i) \tag{2.2}$$

Es gibt verschiedene stetig differenzierbare Funktionen, die als Aktivierungsfunktionen verwendet werden können [16]. Die üblicherweise verwendeten Aktivierungsfunktionen können in zwei Hauptkategorien von Funktionen eingeteilt werden: lineare Aktivierungsfunktion und nicht lineare Aktivierungsfunktion [28]. Im folgenden werden zwei gängige Aktivierungsfunktion vorgestellt .

Rectified Linear Unit (ReLU) ist eine stückweise lineare Aktivierungsfunktion, die am häufigsten von Praktikern und Forschern verwendet wird [28]. Der Erfolg von ReLU basiert auf seiner überlegenen Trainingsleistung gegenüber anderen Aktivierungsfunktionen [15] [18]. Die ReLU-Funktion wird in der Literatur [28] wie folgt definiert:

$$ReLU(x) = max(0,x) \tag{2.3}$$

Die Ableitung von ReLU ist

$$g'(x) = \begin{cases} 0, & \text{if } x < 0 \\ 1, & \text{if } x \geq 0 \end{cases} \tag{2.4}$$

In Abbildung 2 wird die graphische Darstellung von ReLU zeigt. Wenn $x > 0$ ist, ist $g'(x)$ konstant. Ein Vorteil davon ist, das Problem des verschwindenden Gradienten nicht auftreten wird. ReLU hat weitere Vorteile [14]: die Rechnungen des NN sind

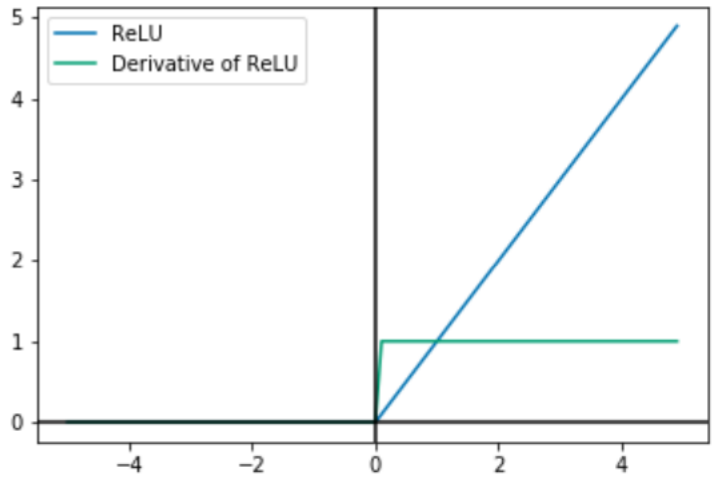


Abbildung 2: Graphische Darstellung der ReLU-Funktion und ihrer Ableitung.

günstiger und das NN mit ReLU konvergiert schneller und im Vergleich zur anderen Aktivierungsfunktionen ist ReLU einfach und Leistungsstark. Der großer Nachteil von ReLU ist, dass alle negativen Werte in Null transformiert werden, was zu toten Neuronen führt. Tote Neuronen haben eine feste Ausgabe von Null. Solche Neuronen zeigen während des Trainings keine Aktivierung und geben keine Informationen weiter. Die Gewichte dieser Neuronen werden nicht aktualisiert.

Die **Sigmoid-Aktivierungsfunktion** ist nichtlinear und hat eine glatte Ableitung. Diese Funktion transformiert den Eingangbereich von $(-\infty, +\infty)$ in den begrenzten Bereich von [0,1]. Sigmoidfunktion ist in der Literatur [28] wie folgt definiert:

$$g(x) = \frac{1}{1 + e^{-x}} \tag{2.5}$$

und die Ableitung dieser Funktion lautet

$$g^{'}(x) = \frac{e^{-x}}{(1 + e^{-x})^2} \tag{2.6}$$

Abbildung 3 zeigt die Sigmoidfunktion und ihre Ableitung. Die Sigmoidfunktion wird aufgrund ihrer Werteverteilung häufig im Ausgabebereich einen neuronalen Netze eingesetzt [9]. Für die Klassifizierung wird Sigmoid oft in Kombination mit anderen Aktivierungsfunktionen verwendet [34]. Der Nachteil der Sigmoidfunktion besteht darin, dass ihre Ableitung zu einem verschwindenden Gradienten führen kann. Nach einem bestimmten Punkt führt dies zu keinem Lernen.

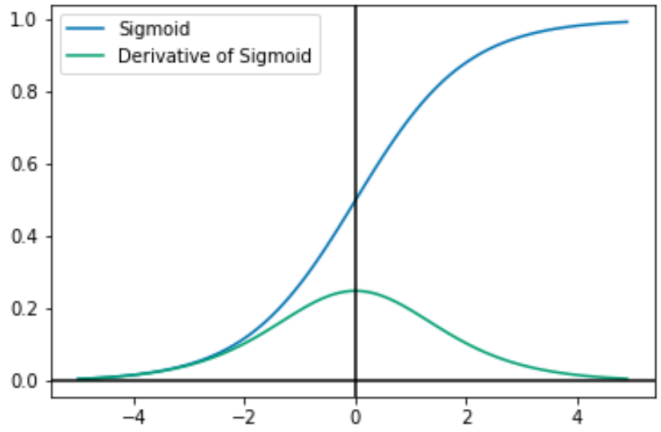


Abbildung 3: Grafische Darstellung der Sigmoid-Funktion und ihrer Ableitung.

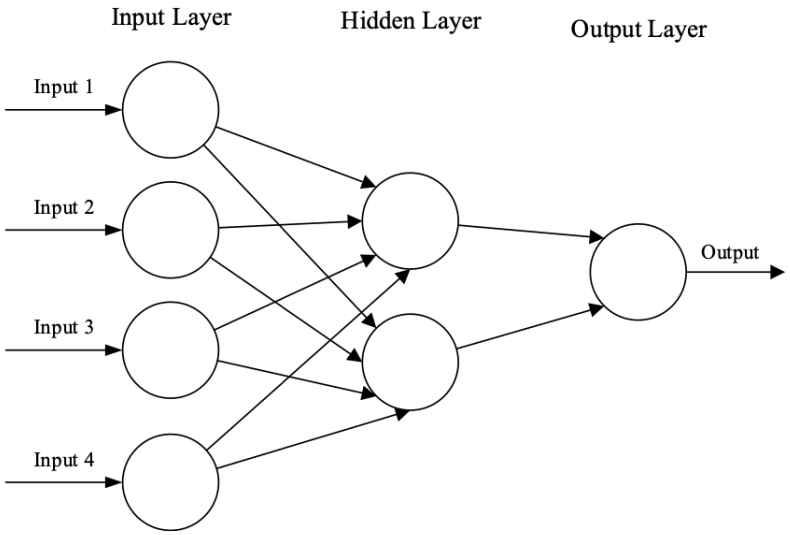


Abbildung 4: Schema eines Drei-Schichte-NNs. [26]

In der Literatur [27] [18] [24] wurden viele verbesserte Versionen von ReLU und anderen Aktivierungsfunktionen vorgestellt. Ein Vergleich der gängigen Aktivierungsfunktionen wurde in Literatur [28] ausführlich eingeführt. Andere Aktivierungsfunktionen werden hier nicht diskutiert.

KNN bestehen aus einer großen Anzahl von rechnerischen Knoten, die miteinander verbunden sind [26]. Eine Schicht (engl. Layer) ist eine Gruppen von Knoten, die die Eingaben verarbeitet, um die Ausgaben zu erzeugen. Die Schichten können je nach der Funktion in drei Arten sortiert werden: Eingabeschicht (engl. Input Layer), versteckte Schicht (engl. Hidden Layer) und Ausgabeschicht (engl. Output Layer). In Abbildung 4 wird ein Drei-Schichten-NN dargestellt. Das Netz besteht aus drei Schichten: einer Ein- und Ausgabeschicht sowie einer versteckten Schicht. Die Anzahl der Ein- und Ausgabe

des Netzes beträgt vier bzw. eins. Die Eingabeschicht nimmt die Eingabedaten für das NN auf und gibt sie an die versteckte Schicht weiter. Die versteckte Schicht ist eine Zwischenschicht im NN und von außen unsichtbar. Die Daten aus der Eingabeschicht werden in der versteckten Schicht verarbeitet. Die Ausgabeschicht dient dazu, die Ergebnisse des NN zu repräsentieren.

NN dürfen nur eine Ein- und Ausgabeschicht haben, aber eine tiefe NN kann mehrere versteckten Schichten haben. Die Anzahl sowie die Verbindung der versteckten Schichten bestimmen die Struktur des neuronalen Netzes [5]. Eine bekannte Netz-Struktur ist **Feedforward-Netz**. Feedforward-Netze sind Netze, bei denen Verbindungen nur zu nächsten Schicht erlaubt sind [41]. Abbildung 4 zeigt beispielsweise ein Feedforward-Netz. Die Struktur dieser Netze ist einfach, aber zusammengesetzt sind sie in der Lage, leistungsstarke Approximationen zu liefern [11]. Manche Feedforward-Netze erlauben sogenannte Shortcut-Verbindungen, welche eine oder mehrere Schichten überspringen [20]. Die Shortcut-Verbindungen dürfen nur in Richtung zur nächsten Schicht verbunden werden.

Beim Training neuronaler Netze werden verschiedene Lernverfahren unterschieden, darunter das überwachte Lernen (engl. supervised learning), das unüberwachte Lernen (engl. unsupervised learning) und das bestärkende Lernen (engl. reinforcement learning). Beim überwachten Lernen werden Trainingsdatensätze benötigt, die Beispiele für Eingaben sowie die Zielwerte für Ausgaben abdecken [19]. Die Ausgaben des Netzes können direkt mit den korrekten Lösungen verglichen werden, um die Netzgewichtungen anhand der Differenz anzupassen [20]. Beim unüberwachten Lernen werden nur Eingabemuster gegeben. Die Ausgabemuster werden aus den vorliegenden Eingabemustern generiert. Das neuronale Netz versucht selbst, ähnliche Eingabemuster zu identifizieren und zu gruppieren. Ein Beispiel hierfür ist die Dimensionsreduktion des Bildes, bei der eine hohe Anzahl von Merkmalen des Bildes in den Daten auf eine niedrigere Anzahl von Merkmalen reduziert wird. Die unüberwachten Lernalgorithmen sollen selbständig wichtige Merkmale aus den Eingangsbildern extrahieren und identifizieren und in dem Netz abbilden. Beim bestärkenden Lernen wird dem Netz nach erfolgten Durchlauf ein Feedback in Form eines Wahrheitswerts gegeben, der definiert, ob das Ergebnis richtig oder falsch ist [20]. Ähnlich wie beim unüberwachten Lernen wird hier keine richtigen Lösungen vorgegeben.

Das Lernprozess oder Training des Netzes ist eine Konfigurierung von Gewichte, um die möglichst genaue Vorhersagen für bestimmte Eingabedaten zu ermöglichen. Dafür muss zunächst der Fehler, also die Unterschiede der Ausgaben zu dem gewünschten Ausgaben für die Eingaben, gemessen werden. Der Fehler wird durch eine Fehlerfunk-

tion berechnet, welche oft auch als Verlustfunktion bezeichnet wird. Viele Funktionen können als Fehlerfunktion verwendet werden. Die Wahl der Fehlerfunktion hängt von der Art des Netzes und dem Problem ab. Das NN kann in der Praxis in zwei Arten unterteilt werden: Regression und Klassifikation. Ein Regressions-NN wird eingesetzt, um kontinuierliche Werte vorherzusagen. Ein Klassifikations-NN wird eingesetz, um Gruppenzugehörigkeiten vorherzusagen. Die Fehlerfunktion kann in zwei Arte sortiert werden. Regression Fehlerfunktionen, die in Regressions-NN verwendet werden z.B. Mean Squared Error (MSE) und Mean Absolute Error (MAE). Klassifikation Fehlerfunktionen, die in Klassifikations-NN verwendet werden z.B. Binary Cross-Entropy. Diese Arbeit handelt sich um eine Regression NN und die Fehlerfunktion MSE würde verwendet. Die MSE ist eine häufig benutzte quadratische Fehlerfunktion. Der Fehler C in MSE wird wie folgenden definiert:

$$C = \frac{1}{n}\sum_{j=1}^{n}(t_j - o_j)^2 \tag{2.7}$$

Dabei bezeichnet n die Anzahl der Eingaben mit denen trainiert wird, t_j die gewünschte Ausgaben und o_j die erlangten Ausgaben. MSE ist aufgrund des quadratischen Terms immer positiv, was bedeutet, dass das Vorzeichen der Differenz keine Rolle spielt. Der quadratische Term kann jedoch sehr groß sein, wenn die Vorhersagen signifikant von den tatsächlichen Werten abweichen In diesem Fall wird MSE signifikant erhört.

Um die Fehlerfunktion zu minimieren, wird das Gradientenabstiegsverfahren verwendet. Im Gradientenabstiegsverfahren wird der lokale Gradient der Fehlerfunktion berücksichtigt und die zugehörigen Parameter werden schrittweise aktualisiert, um den Fehler kontinuierlich zu verringern. Ein weiteres häufig verwendetes Verfahren beim Training neuronaler Netze ist das Backpropagation-Verfahren. Dieses Verfahren versucht, den durchschnittlichen Gesamtfehler zu minimieren, indem es die Gewichte einzeln für jeden Beispiel p ändert [7]. Der Fehler C für Beispiel p ist

$$C_p = \frac{1}{n}\sum_{j=1}^{n}(t_{p,j} - o_{p,j})^2 \tag{2.8}$$

und der Gesamtfehler C ist

$$C = \sum_{p} C_p \tag{2.9}$$

Das Minimum von C kann nicht exakt bestimmt werden, daher wird es durch die Anpassung der Parameter entlang der negativen Steigung der Fehlerfunktion gesucht.

Die Gewichte $w_{i,j}$ von einem Neuron i zu einem Neuron j werden entlang des negativen Gradienten der Fehlerfunktion geändert, bis diese minimal ist. Die Änderung der Gewicht Δw_{ij} wird berechnet durch

$$\Delta w_{ij} = -\eta \sum_p \frac{\partial C_p}{\partial w_{ij}} \tag{2.10}$$

Wobei η ist Lernrate (Learning Rate). Die Geschwindigkeit eines Lernverfahrens ist immer steuerbar von und proportional zu einer Lernrate [20]. Die partielle Ableitung der Fehlerfunktion C ergibt sich durch Verwendung der Kettenregel:

$$\frac{\partial C}{\partial w_{ij}} = \frac{\partial C}{\partial o_j} \frac{\partial o_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}} \tag{2.11}$$

Der erste Faktor wird als Fehlersignal bezeichnet

$$\delta_j = -\frac{\partial C}{\partial o_j} \frac{\partial o_j}{\partial net_j} \tag{2.12}$$

und der zweite Faktor ist

$$\frac{\partial net_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \sum o_j w_{ij} = o_i \tag{2.13}$$

Mit Berechnung von Aktivierungsfunktion wird δ_j wie folgenden berechnet:

$$\delta_j = -\frac{\partial C}{\partial o_j} \frac{\partial \varphi(net_j)}{\partial net_j} = -\frac{\partial C}{\partial o_j} \varphi'(net_j) \tag{2.14}$$

Damit ergibt sich eine vereinfachte Formel für den Backpropagation-Verfahren:

$$\Delta w_{ij} = \eta o_i \delta_j \tag{2.15}$$

mit

$$\delta_j = \begin{cases} \varphi'(net_j)(o_j - t_j) & \textit{falls j Ausgabeneuron ist}, \\ \varphi'(net_j) \sum_k \delta_k w_{jk} & \textit{falls j versteckte Neuron ist} \end{cases} \tag{2.16}$$

Wobei stellt k den Index der Neuronen in der nachfolgenden Schicht zum Neuron j dar.

Zusammengefasst besteht das Backpropagation-Verfahren aus zwei Schritten:

1 Fehler-Berechnung in der Ausgabeschicht und durch Fehler-Rückführung zu den Neuronen der versteckten Schichten

2 Anpassung der Gewichte entgegen des Gradientenanstiegs der Fehlerfunktion

Die Wahl der Lernrate η ist entscheidend für das Training. Wenn eine zu kleine Lernrate

gewählt wird, lernt das Netzwerk langsam. Wenn eine zu große Lernrate gewählt wird, besteht die Gefahr, dass gute Werte übersprungen werden. In Literatur [20] werden gute Werte für die Lernrate im Bereich von $0.01 \leq \eta \leq 0.9$ vorgeschlagen. Um ein neuronales Netzwerk zu trainieren, ist ein Datensatz erforderlich. Je mehr Daten für das Training zur Verfügung stehen, desto besser sind die Ergebnisse des Trainings. Daher ist es wichtig, über eine große Datenmenge zu verfügen, um das Modell effektiv trainieren zu können. Neben die Quantität sollte ein gutes Datensatz auch eine Vielfalt aufweisen, um Verzerrung der Ergebnisse zu vermeiden und das NN auf verschiedene Bedingungen vorzubereiten. Ein Datensatz wird meistens in drei verschiedene Teile aufgeteilt: Trainingsdaten, Validierungsdaten und Testdaten. In der Literatur [35] werden sie wie folgt erklärt: Der **Trainingsdatensatz** wird verwendet, um die Gewichtsanpassung des Algorithmus zu erlernen. Der **Validierungsdatensatz** wird ausschließlich zur Überwachung des Trainingsprozesses verwendet, um diesen zu stoppen, wenn eine optimale Generalisierung erreicht wird. Die **Testdaten** sind von den Trainingsdaten unabhängig und stellt unbekannte Daten dar, um die Fähigkeit eine Netzes zu zeigen.

Je nach Anwendung werden verschiedene Arten von neuronalen Netzwerken entwickelt. In dieser Arbeit würde das CNN benutzt. Das Netz ist eine Sonderform von Feedforward-Netzen und wurden erstmals in Jahr 1998 von LeCun et al. [23] vorgestellt. Es wurde in vielen Forschungsarbeiten gezeigt, dass CNN eine ausgezeichnete Leistung in der Bildverarbeitung aufweist [21] [31] [13]. CNN basiert auf Faltungsoperation und besteht aus drei Arten von Schichten: Convolutional-Layer, Pooling-Layer und Fully-connected-Layer. Zwei Operationen werden in CNN verwendet: Faltung und Pooling. Im Folgenden wird die Funktionsweise von Convolutional-Layer, Pooling-Layer und Fully-connected-Layer sowie Faltung und Pooling erklärt.

Die **Convolutional-Layer** verwendet die Faltungsoperation zwischen den Eingabedaten und einem Filter (Kernel), um Merkmalen in den Eingabedaten zu extrahieren. Der meistens verwendete Filter ist quadratisch. Für das CNN wird die diskrete Faltung im zweidimensionalen verwendet. Für eine Eingabebild x mit einer Größe von b und h, wobei b die Breite und h die Höhe des Bildes sind, und einer Faltungsmatrix mit einer Größe von $f \times f$, wird die diskrete Faltung an der Position (i, j) wie folgt definiert:

$$(x * W)(i,j) = \sum_{m=1}^{f} \sum_{n=1}^{f} x(i+m-a, j+n-a) W(m,n) \tag{2.17}$$

Dabei ist (a,a) der Mittelpunkt von W. Der Wert von a kann berechnet werden als $a = \frac{f+1}{2}$. Die Größe eines Faltungsmatrixs (auch Kernel benennt) beträgt typischerweise

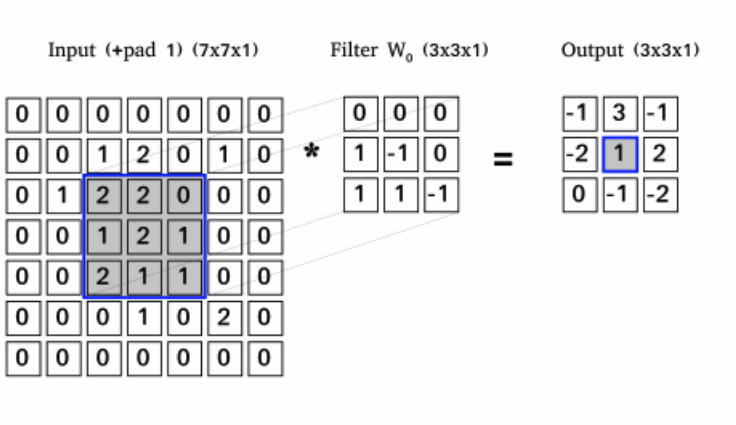


Abbildung 5: Beispiel für eine Faltung mit einer Eingabedatengröße von (5×5) und einem Kernel von (3×3). Nach dem Zero-Padding beträgt die Größe der Eingabedaten von (7×7). Der Kernel gleitet mit einer Stride von zwei über die Eingabedaten. [39]

3 x 3 oder 5 x 5 pixel. Der Kernel wird über die Eingabedaten in bestimmt Schritten verschoben und in jedem Schritt wird ein Frobenius-Skalarprodukt berechnet. Die Dimension der Ausgabedaten kann durch stride und Padding optimiert werden. Stride legt fest, wie weit der Schritt nach jeder Faltung sein wird. Eine größere Stride führt zu einem geringeren Ausgangsvolumen. Es ist notwendig, stride so zu setzen, dass das Ausgangsvolumen eine ganze Zahl ist [2]. Die Informationen an den Rändern des Bildes können durch die Faltung verloren gehen. Zero-Padding kann dieses Problem vermeiden. Dabei werden am Rand der Eingabedaten Nullwerte hinzugefügt, um die Größe der Eingabedaten und der Ausgabedaten beizubehalten. Same-Padding bezieht sich auf ein spezielles Zero-Padding, bei dem die Größe des Ausgabebildes nach der Faltung beibehalten wird. Die Faltungsoperation wird in Abbildung 5 graphisch dargestellt. Die Eingabedaten haben eine Größe von 5×5 Pixeln und der Kernel hat eine Größe von 3×3 Pixeln. Nach Zero-Padding ist die Größe der Eingabedaten von 7×7 Pixeln. Mit einem Stride von 2 ergibt sich eine Ausgabedatengröße von 3×3 Pixeln. Im Allgemeine kann die Größe der Ausgabe wie folgt berechnet werden.

$$\left(\frac{b+2p-f}{s}+1\right) \times \left(\frac{h+2p-f}{s}+1\right) \tag{2.18}$$

Wobei b und h die Breite und Höhe des Bildes darstellen, f die Größe des Kenels, s den Stride und p das Padding angibt. Die Größe von p ist wählbar und wird normalerweise als $p = \frac{f-1}{2}$ berechnet. Bei No-Padding ist $p = 0$. Für das Beispiel in Abbildung 5 berechnet sich die Größe der Ausgabedaten wie folgt: $\left(\frac{5+2\cdot1-3}{2}+1\right) \times \left(\frac{5+2\cdot1-3}{2}+1\right) = (3 \times 3)$.

Das **Pooling Layer** zielt darauf ab, die Dimension der Daten zu reduzieren, um die

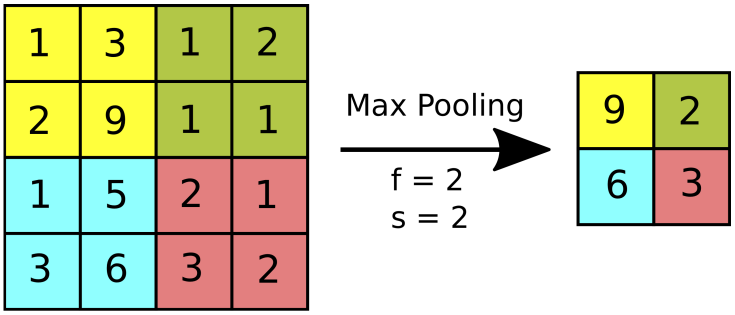


Abbildung 6: Beispiel für Max-Pooling. Die Größe des Kernels f beträgt (2×2) und der Stride s ist 2 [30].

Anzahl der Parameter und die Rechenkomplexität des Modells weiter zu verringern [26]. Pooling ähnelt der Faltungsoperation. Max-Pooling ist in CNN die meisten verwendete Pooling-Methode. Es teilt das Bild in Unterbereiche (Rechtecke) auf und gibt nur den maximalen Wert im Inneren dieses Unterbereichs zurück [3]. Eine häufig verwendete Größe des Kernels für das Max-Pooling ist (2×2) mit einer Stride von 2 [3]. Da für Max-Pooling kein Zero-Padding verwendet wird, also $p = 0$, kann die Größe der Ausgabedaten wie folgt berechnet werden:

$$\left(\frac{b-f}{s}+1\right) \times \left(\frac{h-f}{s}+1\right) \tag{2.19}$$

Ein Beispiel für Max-Pooling wird in Abbildung 6 dargestellt. Die Größe der Eingabedaten und des Kernels beträgt (4×4) und (2×2). Es wird ein Stride von $s = 2$ verwendet.

Fully-connected-Layer ist ähnlich angeordnet wie Neuronen in einem traditionellen KNN. Die Neuronen im Fully-connected-Layer sind direkt mit den Neuronen in den beiden benachbarten Schichten verbunden [26]. Die Ausgabedaten der Convolutional- und Pooling-Layer werden zuerst ausgerollt. Die Fully-connected-Layer nimmt die ausgerollten Daten an und gibt die endgültigen Wahrscheinlichkeiten für jedes Label an. Der Hauptnachteil einer Fully-Connected-Schicht besteht darin, dass sie viele Parameter enthält, die komplexe Berechnungen beim Training erfordern [3].

2.2 Grundlagen des Autoencoder

Ein häufig in der Bildverarbeitung verwendetes neuronales Netzwerk ist der Autoencoder (AE). Er wird beliebt im Unsupervised-Learning eingesetzt. Aufgrund seiner Struktur eignet sich ein Autoencoder besonders gut für die Untersuchung der Arithmetik im latenten Raum. In diesem Abschnitt werden die Grundlagen des AE erläutert.

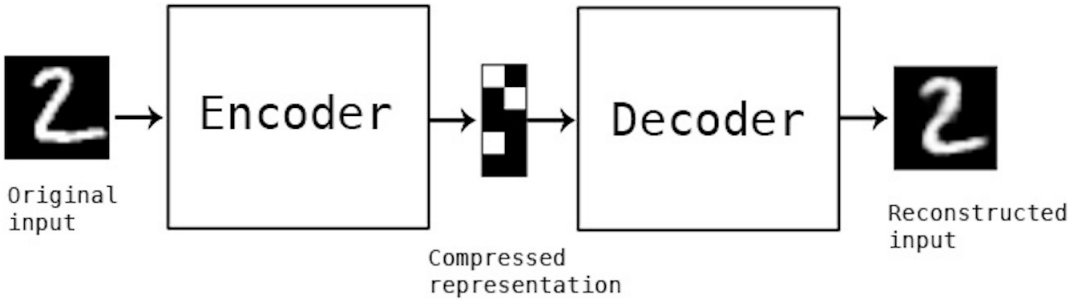


Abbildung 7: Darstellung der Architektur von Autoencoder. Das Eingangsbild wird in eine komprimierte Darstellung kodiert und anschließend decodiert.

AE ist ein spezielles neuronales Netz, das erstmals in [22] eingeführt wurde. Der AE wird entwickelt, um die Eingabe in eine komprimierte und sinnvolle Darstellung zu kodieren und sie anschließend so zu decodieren, dass die rekonstruierte Eingabe den Originaldaten so ähnlich wie möglich ist. Um dies zu erreichen, besteht ein AE aus zwei wichtige Modulen, dem Encoder und dem Decoder. Die Dimension der kodierten Daten ist in der Regel niedriger als die der originalen Eingabedaten. Die kodierten Daten, auch als latente Darstellung bezeichnet, befinden sich zwischen dem Encoder und dem Decoder. Der niedrigdimensionale Raum, in dem die latente Darstellung liegt, wird als latenter Raum bezeichnet. Ein Vorteil der AE-Architektur besteht darin, dass die Daten im latenten Raum direkt von außen zugänglich sind. Dadurch ist es einfach, arithmetische Operationen auf den latenten Daten durchzuführen. Eine allgemeine Struktur des AE wird in Abbildung 7 dargestellt.

In Literatur [1] würde das mathematische Modell des AE wei folgt dargestellt. Der Encoder g_{w_E}, der aus einer oder mehreren neuronalen Schichten besteht, erhält eine Eingabe x und gibt die entsprechende niedrigdimensionale latente Darstellung z zurück. Dies kann wie folgt dargestellt werden:

$$z = g_{w_E}(x) \tag{2.20}$$

Wobei w_E repräsentiert die Gewicht des Encoders.

Die latente Darstellung z wird in den Decoder f_{w_D} eingespeist. Der Decoder lernt, die Ausgabe $\widetilde{x}$ zu erzeugen, indem er die ursprüngliche Eingabe rekonstruiert. Dies kann wie folgt dargestellt werden:

$$\widetilde{x} = f_{w_D}(z) \tag{2.21}$$

Wobei w_D repräsentiert die Gewicht des Decoders.

Der AE wird trainiert, um die Parametern w_E und w_D sowohl für den Encoder als auch für den Decoder zu optimieren. Dadurch wird der Rekonstruktionsfehler minimiert, der den Unterschied zwischen der tatsächlichen Eingabe und der rekonstruierten Eingabe misst. Der vorher erklärte MSE ist eine gängige Funktion zur Berechnung des Rekonstruktionsfehlers in AE [1]. Der Rekonstruktionsfehler kann mit Hilfe von MSE definiert als:

$$C(w_E, w_D) = \frac{1}{n}\sum^{n}(x - f_{w_D}(g_{w_E}(x)))^2 \tag{2.22}$$

Hier repräsentiert das Symbol n die Anzahl der Trainingsdaten.

Die Dimension der latenten Darstellung z sollte immer kleiner sein als die der Eingabedaten x. Eine niedrigdimensionale latente Darstellung, auch als Bottleneck bezeichnet, ermöglicht es dem AE, nur wichtige, unterscheidende und wertvolle Merkmale der Eingabe zu lernen. Diese Reduktion der Dimensionen ist eine grundlegende Funktion eines AE. Daher wird AE häufig zur Merkmalsextraktion und Datenkompression verwendet. Eine weitere Anwendung von AE besteht in der Datenmanipulation im latenten Raum. Dabei können gezielte Veränderungen an den gelernten Merkmalen der Eingabe vorgenommen werden.

2.3 Convolutional-Autoencoder und ResNet-Autoencoder

Um einen Leistungsstarken AE zu entwickeln, wird er mit verschiedenen Architekturen des neuronalen Netzwerks aufgebaut. In der Literatur wurden verschiedene Typen von Autoencodern vorgestellt und diskutiert, wie zum Beispiel Variational Autoencoder (VAE) [25], Convolutional Autoencoder (CAE) [36], Sparse Autoencoder (SAE) [44] und ResNet Autoencoder (RAE) [41]. In dieser Arbeit würde ein neuronale Netze entwickelt, es sich um Convolutional- und ResNet-Autoencoder handelt. Die zwei Variante des AE werden im diesem Abschnitt erklärt.

CAE wird mit der CNN als Baustein sowohl für den Encoder als auch für den Decoder aufgebaut. Es bedeute, dass der Encoder und Decoder mithilfe von Convolutional-Layer implementiert werden. Der Encoder umfasst convolutional-Layer und Max-Pooling-Layer, während der Decoder aus Deconvolution-Layer und Upsampling-Layer besteht. Deconvolution und Upsampling sind die Umkehroperationen von Faltung und Max-Pooling. Ein Beispiel für CAE wird in Abbildung 8 dargestellt. Die Dimension den Eingabedaten von $400 \times 400 \times 3$ wird durch die Convolutional-Layers auf $25 \times 25 \times 5$ reduziert. Der Encoder und Der Decoder haben jeweils drei Convolutional-Layers. Nach

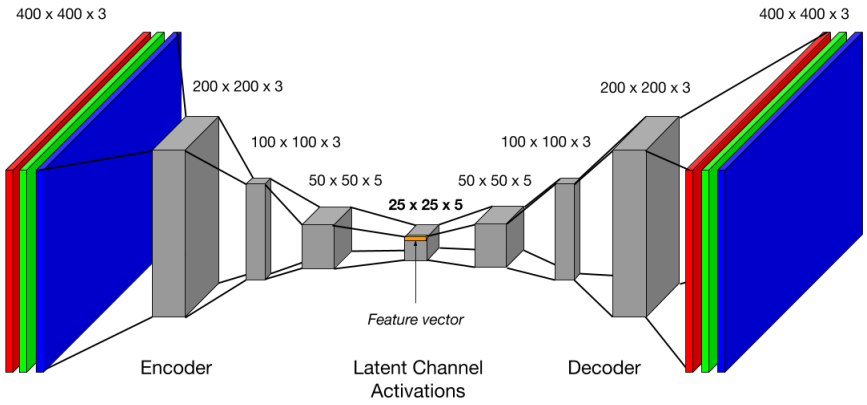


Abbildung 8: Struktur eines Convolutional Autoencoder [32]. Der Encoder-Teil und der Decoder-Teil haben jeweils drei Convolutional Layers.

einer symmetrischen Bearbeitung im Decoder wird ein rekonstruiertes Bild generiert. CAE kombiniert die Vorteile von CNN und AE und bietet eine verbesserte Leistung im Vergleich zu traditionellen AE, insbesondere bei der Verarbeitung von zweidimensionale Daten [6]. Im Vergleich zu CNN werden CAE ausschließlich darauf trainiert, Filter (Der Kernel wird in AE auch als Filter bezeichnet) zu lernen, um die Merkmale des Eingabedaten zu extrahieren und sie zu rekonstruieren [1].

Tiefe Convolutional Neural Networks [21] haben zu einer Reihe von Durchbrüchen bei der Bildverarbeitung geführt [43] [33]. Wenn ein NN mehrere Schichten hat, tritt das Problem des verschwindenden Gradienten (Vanishing Gradient) auf. Das bedeute, dass wenn das Netzwerk tiefer wird, seine Leistung sättigen oder schnell abnehmen kann [42]. Um das Problem zu lösen, wurde das Residuale lernen in der Literatur [17] eingeführt. Das residuale Neuronale Netz wird als ResNet bezeichnet. In ResNet werden die Shortcut-Verbindung verwendet. Ein ResNet-Autoencoder ist ein Autoencoder, der die Shortcut-Verbindung enthält. Eine Residualeinheit wurde in der Literatur [17] eingeführt. In Abbildung 9 wird Die Residualeinheit dargestellt. Sie besteht aus zwei Schichten und einer Short-Cut-Verbindung, die zwei Schichten überspringt. Das Symbol $\oplus$ in Abbildung 9 steht für die elementweise Addition. Die Ausgabe der Residualeinheit kann berechnet als:

$$o_3 = o_2 + x \tag{2.23}$$

Wobei x und o_3 die Eingabe und Ausgabe der Residualeinheit darstellen, während xo_1 und o_2 die Ausgaben von Schicht 1 und Schicht 2 repräsentieren. Es sollte beachtet werden, dass die Dimension von o_3 und x die gleiche Größe haben müssen. Die Auswahl der Schicht innerhalb der Residualeinheit ist flexibel. In dieser Arbeit würde den Convolutional-Layer benutzt.

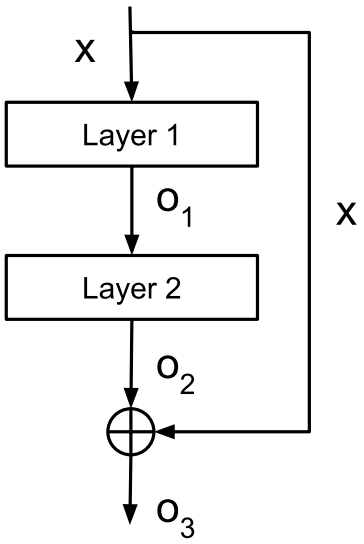


Abbildung 9: Residualeinheit mit zwei Schichten.

In Literatur [41] wurde ein C-RAE vorgestellt. Das Autoencoder basieren auf dem CAE und integriert Shortcut-Verbindungen. Die Struktur des C-RAE wird in Abbildung 10 dargestellt. Ähnlich wie beim AE besteht ein C-RAE aus dem Encoder und dem Decoder. Eine Shortcut-Verbindung verbindet den Eingabe des Encoders mit der Ausgabe des Encoders. Die Shortcut-Verbindung überspringt mehrere Convolutional- und Max-Pooling-Layers. Diese Architektur des C-RAE kombinieren die Vorteile von CAE und ermöglichen es, tiefe Netzwerke zu entwerfen, ohne die Leistung des Merkmalslernens beeinträchtigt wird. Der Experiment in [41] zeigt, dass C-RAE im Vergleich zum CAE eine deutlich geringere Verschlechterung der Klassifikationsgenauigkeit aufweist und eine höhere Genauigkeit bei der Merkmalsextraktion erzielt.

Ähnlich wie beim AE kann die Ausgabe des C-RAE als $f_{W_D}(g_{W_E}(x))$ definiert werden. Die Gewichtsmatrix, die der Schicht *l* entspricht, kann als $W^{(l)}$ bezeichnet werden. Für den Encoder kann die versteckte Darstellung $v^{(l)}$ in der versteckten Schicht *l* wie folgt berechnet werden:

$$v^{(l+1)} = r(v^{(l)}) + g(v^{(l)}) \tag{2.24}$$

Der Fehler des C-RAE kann durch den MSE berechnet werden:

$$C_\omega = \frac{1}{n}\sum_{i=1}^{n}(x_i - o_i)^2 \tag{2.25}$$

Die Variable ω repräsentiert die Parameter des Autoencoders, einschließlich Gewichten und Biases.

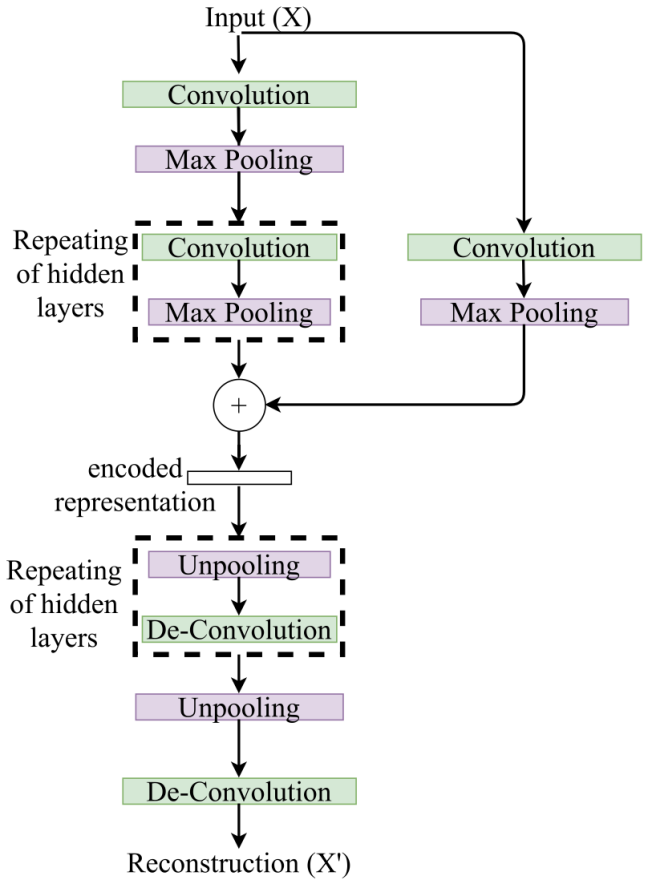


Abbildung 10: Struktur eines Convolutional-ResNet-Autoencoder.

Das Training des RAE kann auch durch Anwendung des Backpropagation-Verfahrens zur Minimierung der Fehlerfunktion erfolgen. Die pseudo-code in Algorithm 1 wird ein Beispiel für Training des RAE dargestellt.

2.4 SSIM und Sobel-Filter

Der MSE ist eine am weitesten verbreitete Qualitätsmetrik, da sie einfach zu berechnen ist und eine klare physikalische Bedeutung hat [40]. Allerdings ist der MSE nicht besonders gut auf die wahrgenommene visuelle Qualität abgestimmt. Daher würde structural similarity index (SSIM) entwickelt, um die Bildqualität zu bewerten. Der SSIM vergleicht lokale Muster von Pixelintensitäten, die für Helligkeit und Kontrast normalisiert wurden [40]. Er berechnet die Strukturelle Ähnlichkeit zwischen zwei Bildern unter Berücksichtigung von Helligkeit und des Kontrast. Der SSIM wird auf verschiedenen Fenstern eines Bildes Hilfe von kleinen Bildfenster berechnet. In dieser Arbeit wurde der SSIM als ergänzende Metrik verwendet. Die ursprüngliche Formel [40] für den SSIM

Algorithm 1 RAE Training

Require: Training set X
Ensure: Trained RAE

Random Weight initialization
for each epoch e **do**
  for $i = 1 ... n$ **do** ▷ number of training samples
    $x_i \leftarrow$ pick random input record from X
    $v \leftarrow x_i$
    for $l = 1 ... L$ **do** ▷ each hidden layer l in encoder
      $v_e \leftarrow v$
      for $j = 1 ... E$ **do** ▷ each layer j in e
        $v_e \leftarrow \varphi(W^{l,j} v_e + c^{(l,j)})$
      end for
      $v \leftarrow W_r^{(l)} v + v_e$ ▷ add residual connection to the hidden activation v_e
    end for
    $o_i \leftarrow v$
    for $l = 1 ... L$ **do** ▷ each hidden layer l in decoder
      $o_i \leftarrow \varphi(U^{(l)} o_i + d^{(l)})$
    end for
  end for
  Compute the reconstruction loss: $C_\omega = \frac{1}{n} \sum_{i=1}^{n} (x_i - o_i)^2$
  Perform One-step of the optimizer: $\omega = argmin_\omega(C_\omega)$
end for

lautet:

$$SSIM(x,y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)} \tag{2.26}$$

mit:

- Zwei Fenstern x und y mit einer Größe von $N \times N$,
- μ_x und μ_y den Mittelwert von x und y,
- σ_x^2 und σ_y^2 die Varianz von x und y,
- σ_{xy} die Kovarianz von x und y,
- $c_1 = (k_1 L)^2$, $c_2 = (k_2 L)^2$ zwei Variablen, um die Division mit einem kleinen Nenner zu stabilisieren,
- L der Dynamikbereich der Pixelwerte (typischerweise ist $2^{\#bits\ per\ pixel} - 1$)
- $k_1 = 0{,}01$ und $k_2 = 0{,}03$ standardmäßig

Um eine bessere Kantenrekonstruktion für ein C-RAE zu erreichen, werden in dieser Arbeit zusätzliche Kanteninformationen, die durch einen Sobel-Filter geliefert werden, in das Netz eingespeist. Die meisten Kantendetektion-Methoden arbeiten unter der Annahme, dass eine Kante dort auftritt, wo es ein sehr steiles Intensitätsgefälle im Bild gibt. Die Gradientenmethode wird oft bei der Kantendetektion verwendet. Sie erkennt Kanten, indem sie nach dem Maximum und Minimum in der ersten Ableitung des Bildes sucht [37]. Der Sobel-Filter ist eine Gradientenmethode für Kantendetektion. Zwei Faltung-Kenel von (3×3) für die Richtung X und Y wird gegeben als:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

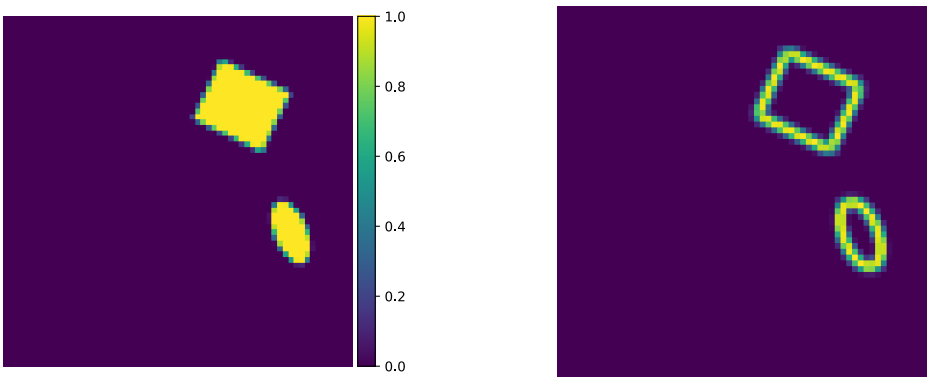
Ein Kernel ist einfach der andere um 90 Grad gedreht. Der Operator berechnet den Gradienten der Bildintensität an jedem Punkt, indem er die Richtung des größtmöglichen Anstiegs von hell zu dunkel und die Änderungsrate in dieser Richtung angibt [37]. Der absolute Betrag G des Gradienten kann berechnet werden als:

$$G = \sqrt{G_x^2 + G_y^2} \tag{2.27}$$

Typischerweise wird eine Näherung des Betrags mit folgender Formel berechnet [4]:

$$|G| = |G_x| + |G_y| \tag{2.28}$$

Im Vergleich zum anderen Kantendetektion hat Sobel-Filter zwei Vorteile [12]: Er hat einen gewissen Glättungseffekt auf das Rauschen im Bild. Die Elemente der Kante auf beiden Seiten wurden verstärkt, so dass die Kante dick und hell erscheint. Abbildung 11 zeigt ein Beispiel eines Datensatzes und seines Kantenbildes nach Anwendung des Sobel-Filters Das Bild auf der linken Seite (siehe Abbildung 11a) ist das originale



(a) Das originale Bild

(b) Graphische Darstellung der Kanteninformationen

Abbildung 11: Darstellung eines originalen Bildes (links) und seinen Kanteninformationen (rechts). Die Bilder haben einen Wertebereich von 0 bis 1.

Bild. Für die manuelle Bewertung der Bilder wird in dieser Arbeit die standardmäßige Farbkarte *viridis* von Matplotlib verwendet. In der *viridis* Farbkarte werden die Werte 1 und 0 durch die Farben Gelb und Lila dargestellt. Die Kanteninformationen des originalen Bildes werden in Abbildung 11b graphisch dargestellt.

3 Konzept und Implementierung

Das Konzept des neuronalen Netzes und die Implementierung des ResNet-Autoencoders werden in diesem Kapitel erläutert. In Abschnitt 3.2 wird die Struktur des neuronalen Netzes erklärt, während in Abschnitt 3.1 die Erstellung des Datensatzes erläutert wird. Eine detaillierte Beschreibung der Implementierung des ResNet-Autoencoders findet sich in Abschnitt 3.3.

3.1 Erstellung von Trainings- und Testdatensätzen

Wie in Abschnitt 2.1 erklärt, wird ein Datensatz für das Training des ResNet-Autoencoders benötigt. Der vorhandene Datensatz für Bilder (z.B. das MNIST-Dateset) ist jedoch zu komplex für diese Arbeit. Deshalb wurden zuerst die Datensätze für das Training generiert. Jedes generierte Bild enthält nur ein oder zwei Objekte, entweder Ellipsen oder Rechtecke. Ein Sample aus dem Datensatz enthält drei Bilder. Zwei dieser Bilder enthalten jeweils ein Objekt. Das dritte Bild ist eine Kombination aus den anderen beiden Bildern, bei der die Objekte übereinandergelegt wurden. In Abbildung 12 wird ein Sample aus drei Bilden dargestellt. Der Bildvektor wird im Wertebereich von 0 bis 1 dargestellt. Dabei steht die Farbe Gelb für den Wert 1 und die Farbe Lila für den Wert 0. Der Vektor eines farbigen Bildes ist dreidimensional: Zwei Dimensionen sind die Größe des Bildes (Breite und Höhe), und die dritte Dimension ist der Farbkanal (in der Regel RGB, also Rot, Grün und Blau). In dieser Arbeit werden alle Bilder in Graustufen erstellt, was bedeutet, dass es nur einen Farbkanal gibt. Zum Training des Netzes werden 10.000 Samples in dieser Arbeit verwendet.

Die Funktion *'draw_samples( )'* wurde in der Programmiersprache Python geschrieben, um die Datensätze zu generieren. Dabei können die Parameter in der Funktion ange-

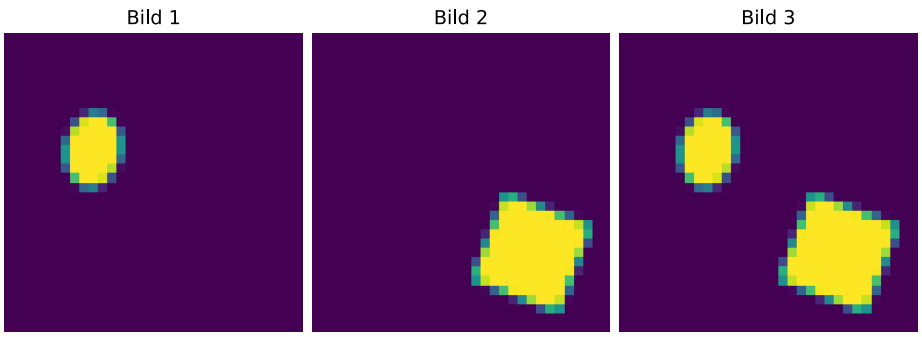


Abbildung 12: Ein Beispiel-Sample mit einer Größe von 32 $\times$ 32 Pixeln.

passt werden, um unterschiedliche Datensätze zu erzeugen. Die Parameter, die vom Benutzer angegeben werden, sind die Größe des Bildes, die Größe des Objekts im Bild, die Überlappung, das Überschneidungsverhältnis und das Hintergrundrauschen. Die Bilder sollen quadratisch sein, was bedeutet, dass Breite und Höhe des Bildes identisch sind. In dieser Arbeit werden Bilder mit einer Größe von 32×32 und 64×64 erstellt. Weil die Objekte um 360 Grad um ihren Mittelpunkt gedreht werden können, wird die Größe des Objekts auf einen quadratischen Bereich begrenzt. Die Größe des Objekts wird in dieser Arbeit zwischen drei und acht Pixeln begrenzt. Dies wird wie folgt genau bestimmt. Zunächst werden die minimalen und maximalen Größen angegeben. Dann werden die Längen beiden Kanten eines Rechteckes sowie die Längen der Hauptachse und Nebenachse einer Ellipse durch die Zufallszahlmethode *'np.random.default_rng.integers( )'* im begrenzten Bereich festgelegt. Am Ende gibt die Zufallszahlmethode zufällige Ganzzahlen aus der "diskreten uniformen"Verteilung des angegebenen Datentyps zurück. Der Parameter *Überlappung* entscheidet, ob sich zwei Objekte im dritten Bild überlappen oder nicht. Es wird vereinfacht, dass die Überlappung durch die Bounding-Box der Objekte berechnet wird, um das Überlappungsverhältnis zu bestimmen. Das Überschneidungsverhältnis wird in dieser Arbeit wie folgt definiert:

$$\textit{Overlapped Ratio} = \frac{\textit{Overlapped Area}}{\textit{Area of the smallest Bounding Box}} \tag{3.1}$$

Der Parameter *Hintergrundrauschen* ist eine Erweiterung für die Erstellung von Datensätzen. Mit der Funktion *'draw_samples( )'* können Bilder mit Hintergrundrauschen erzeugt werden. Das Hintergrundrauschen wird dabei gemäß einer White-Gaussian-Verteilung mit festgelegten Erwartung (0,3) und Varianz (0,05) generiert. Die Position des Objektes hängt von die Größe des Objektes ab, um das gesamte Objekt auf einem Bild darzustellen.

Das Ablaufdiagramm für die Generierung eines Sample wird in Abbildung 13 dargestellt. Zunächst werden zufällige Zahlen für die Größe und den Rotationswinkel des Objekts generiert. Mit diesen Parametern wird dann eine Bounding-Box für das Objekt erstellt. Die Bounding-Box wird verwendet, um zu überprüfen, ob sich die Objekte überlappen und wie groß das Überlappungsverhältnis ist. Wenn beispielsweise eine Überlappung erlaubt ist und das Überlappungsverhältnis kleiner als 0,5 ist, wird das Objekt generiert. Andernfalls müssen neue Zufallszahlen generiert und die Bounding-Box erneut überprüft werden. Welches Objekt im Bild 1 oder Bild 2 erzeugt wird, wird ebenfalls zufällig bestimmt. Es ist möglich, dass ein Sample nur eine Ellipse oder ein Rechteck enthält, was die Vielfalt des Datensatzes erhöht.

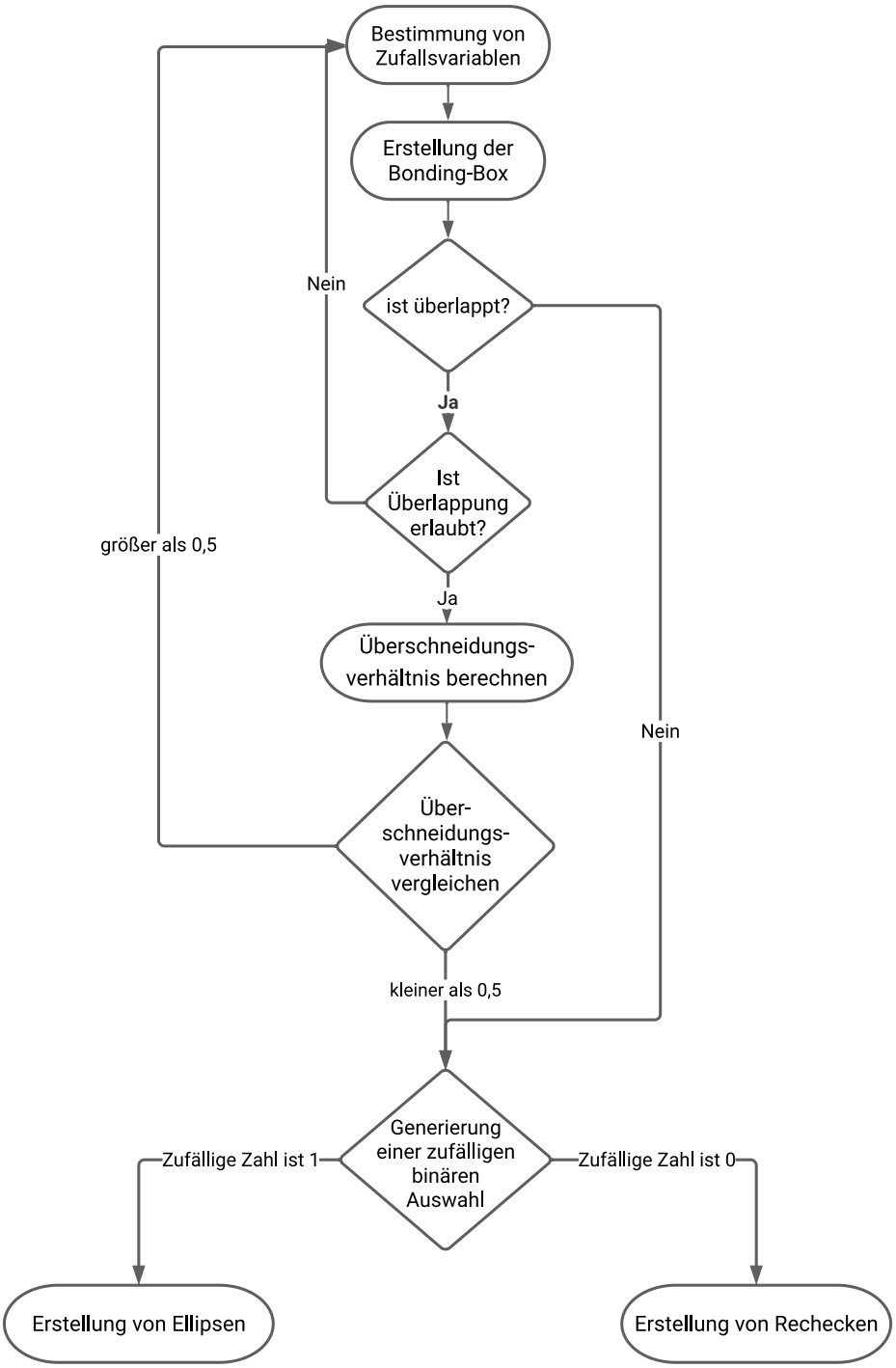


Abbildung 13: Ablaufdiagramm für Generierung eines Samples

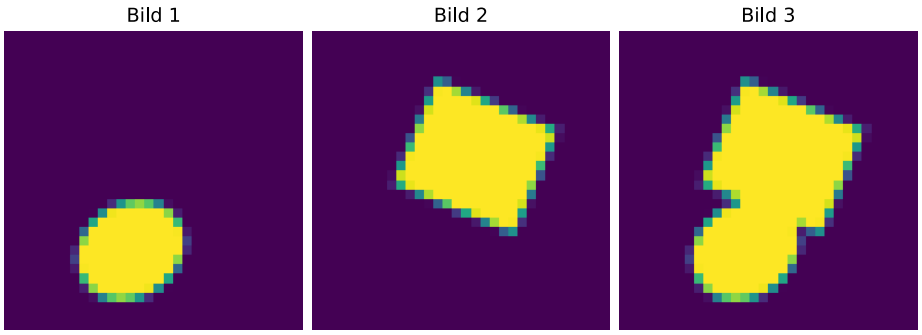


Abbildung 14: Ein Beispiel-Sample mit Überlappung. Die Größe des Bildes beträgt 32×32 Pixel.

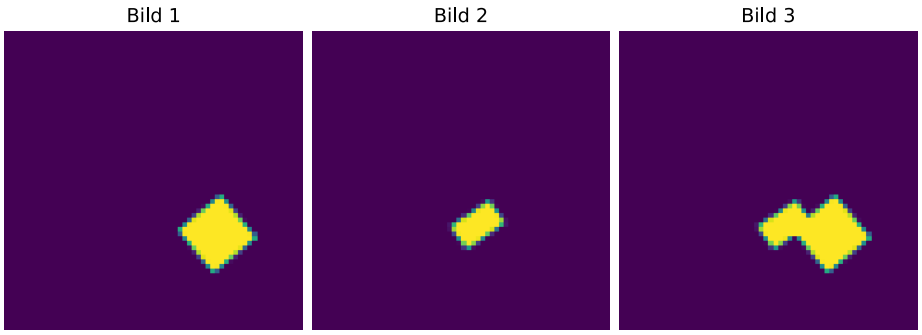


Abbildung 15: Ein Beispiel-Sample mit Überlappung. Die Größe des Bildes beträgt 64×64 Pixel.

Abbildung 14 zeigt ein Sample mit einer Größe von 32×32 Pixeln. Das dritte Bild enthält zwei überlappende Objekte. Die beiden Objekte sind relativ groß und es gibt im Bild wenig freien Platz. Es würde schwierig sein, mehr als zwei große Objekte auf einem Bild darzustellen. Daher wurde ein Datensatz mit größeren Bildern für weitere Untersuchungen erstellt. In Abbildung 15 wird ein Sample mit der Bildgröße von 64×64 Pixeln dargestellt. Das Sample enthält ein großes und ein kleines Rechteck, die sich leicht überlappen.

3.2 Struktur des neuronalen Netzes

Zu Beginn soll das Konzept des in dieser Arbeit verwendeten neuronalen Netzes erklärt werden. Das neuronale Netz basiert auf dem Autoencoder und ist darauf ausgelegt, zwei Bilder zu addieren. Eine anschauliche Darstellung des Konzepts des neuronalen Netzes ist in Abbildung 16 zu finden.

Das neuronale Netz besteht aus drei Teilen: dem Encoder, dem latente Raum und dem

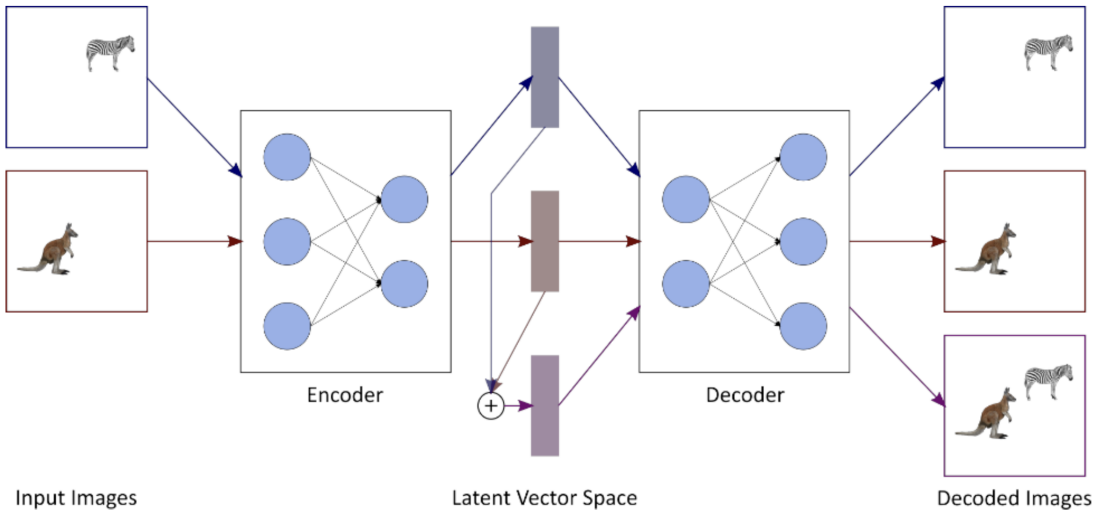


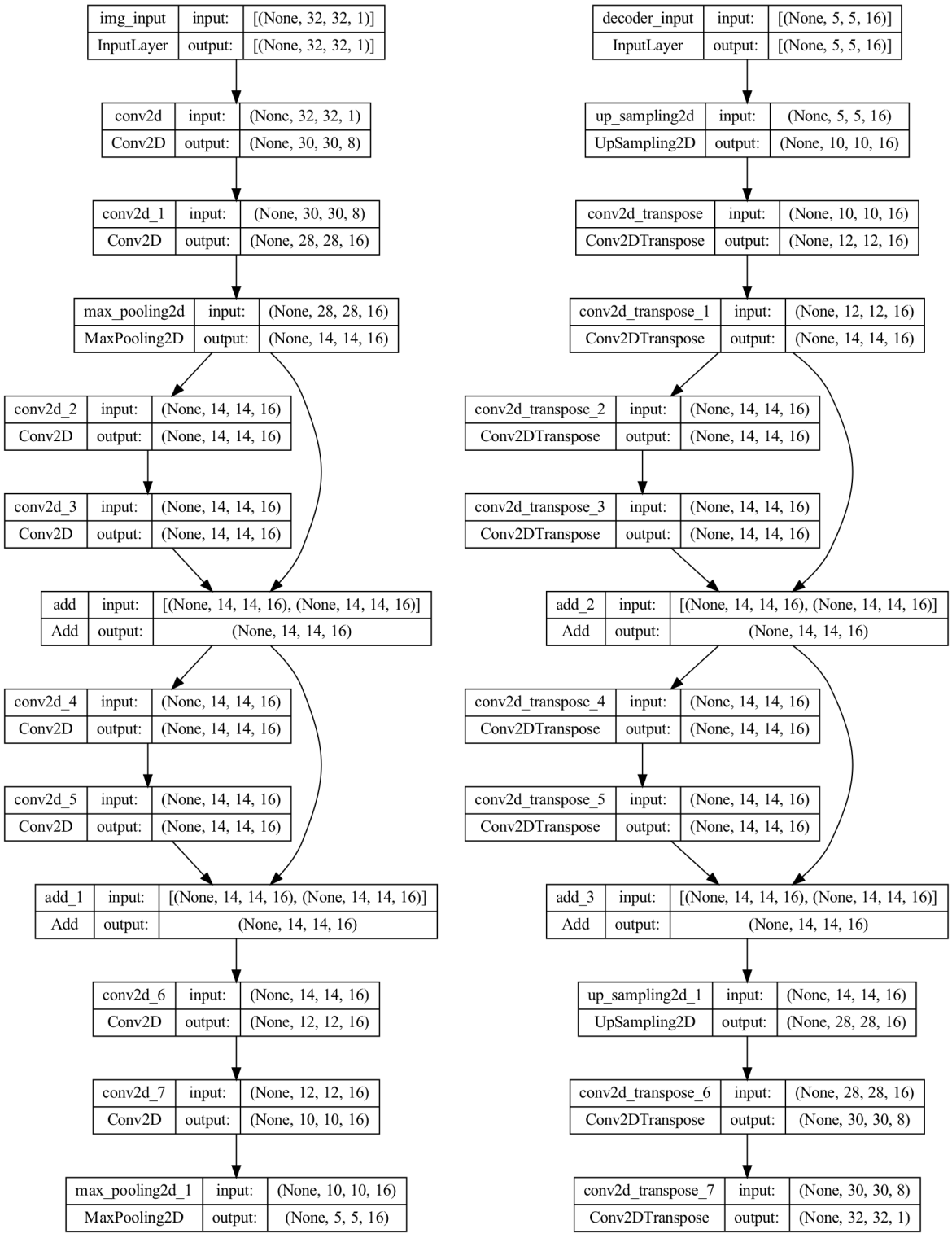
Abbildung 16: Das Konzept des neuronalen Neztes

Decoder. Der latente Raum befindet sich zwischen dem Encoder und dem Decoder und wird für die arithmetischen Operationen genutzt. Zwei Bilder mit unterschiedlichem Inhalt (in diesem Diagramm ein Zebra und ein Känguru) werden dem Autoencoder als Eingabedaten übergeben. Der Encoder transformiert die Bilder in zwei komprimierte Daten und gibt sie im latenten Raum aus. Die beiden Daten sind sogenannte latente Vektoren und enthalten jeweils die Merkmale der beiden Eingangsbilder. Im latenten Raum werden die beiden Vektoren addiert, um die Merkmale beider Bilder zu überlagern. Nach der Berechnung gibt es nun einen zusätzlichen, dritten latenten Vektor im latenten Raum. Der Decoder verwendet diese drei Vektoren, um drei Bilder als Ausgangsbilder zu rekonstruieren. Zwei der rekonstruierten Bilder sollen den Originalbildern ähnlich sein. Das dritte Ausgangsbild soll alle Merkmale der beiden Originalbilder enthalten, was bedeutet, dass beide Eingangsbilder im dritten Ausgangsbild kombiniert werden sollen. Als Beispiel in Abbildung 16 soll das Netz zwei Ausgangsbilder erzeugen, die jeweils ein Zebra und ein Känguru zeigen, sowie ein drittes Bild, das sowohl das Zebra als auch das Känguru enthält. Analog zur Addition kann auch die Subtraktion im latenten Raum durchgeführt werden. Bei der Subtraktion im latenten Raum werden zwei Bilder als Eingabe übergeben, wobei eines ein Zebra und ein Känguru enthält und das andere nur ein Känguru zeigt. Nach der Subtraktion soll im dritten Ausgangsbild nur das Zebra enthalten sein.

3.3 Implementierung des ResNet-Autoencoders

Der Autoencoder wird in der Programmiersprache Python implementiert. Für die Implementierung wurden das Framework TensorFlow sowie die Bibliotheken Keras verwendet. TensorFlow ist eine umfassende Open-Source-Plattform für maschinelles Lernen. Nach der Veröffentlichung im Jahr 2015 hat TensorFlow in der KI-Gemeinschaft große Popularität erlangt. TensorFlow ist modular aufgebaut und ermöglicht die Entwicklern die komplexe neuronale Netze schnell zu erstellen und zu trainieren. Keras ist eine Open-Source-basierte Bibliothek, die auf Maschinelles-Lernen-Framework wie TensorFlow einsetzbar ist. Die Bibliothek stellt ein High-Level-API zur Verfügung, sodass der Zeitaufwand für die Programmierung von NN stark reduziert wird.

Der ResNet-Autoencoder würde durch die Functional API von Keras implementiert. Zuerst werden der Encoder und Decoder definiert. Wie bereits im Abschnitt 2.3 erläutert, basiert der in dieser Arbeit verwendete ResNet-Autoencoder auf dem Convolutional-Autoencoder und verwendet Residualverbindungen. Der Encoder wird in Abbildung 17a schematisch dargestellt und besteht aus zwei residuellen Blöcken, die jeweils aus zwei Convolutional-Schichten, einer Additionsschicht und einer Residualverbindung bestehen. Beide Convolutional-Schichten sind mit Same-Padding eingestellt, um die Dimensionen der Daten innerhalb der residuellen Blöcke beizubehalten und somit eine problemlose Addition mit einer Residualverbindung zu ermöglichen. Zudem werden zwei MaxPooling-Schichten verwendet, um die Dimensionen der Daten üblicherweise zu halbieren. Zusätzlich zu den residuellen Blöcken werden vier weitere Convolutional-Schichten eingesetzt. Die Struktur des Decoders ist symmetrische zur des Encoder und wird in Abbildung 17b dargestellt. Im Decoder werden transponierte Convolutional-Schicht und UpSampling-Schicht verwendet. Die detaillierten Einstellungen aller Schichten im Encoder und Decoder sind in Tabelle 1 aufgeführt.



(a) Die Struktur des Encoders

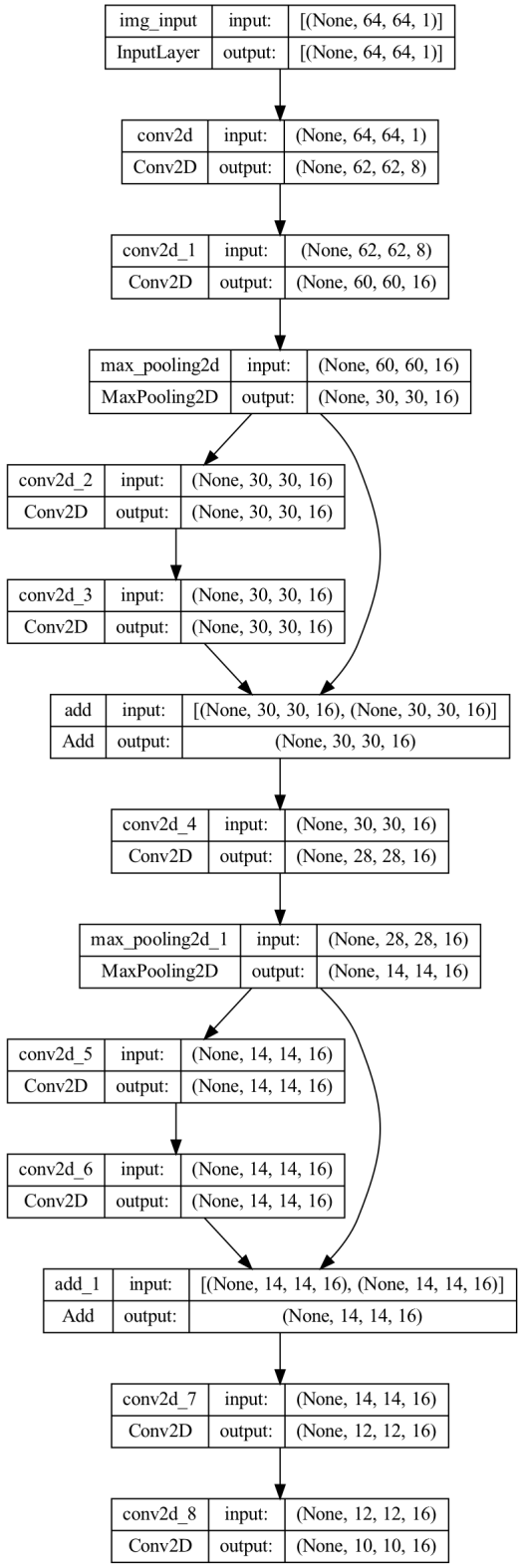
(b) Die Struktur des Decoders

Abbildung 17: Die Struktur des Encoders und Decoders für eine Eingangsbild von 32×32 Pixeln.

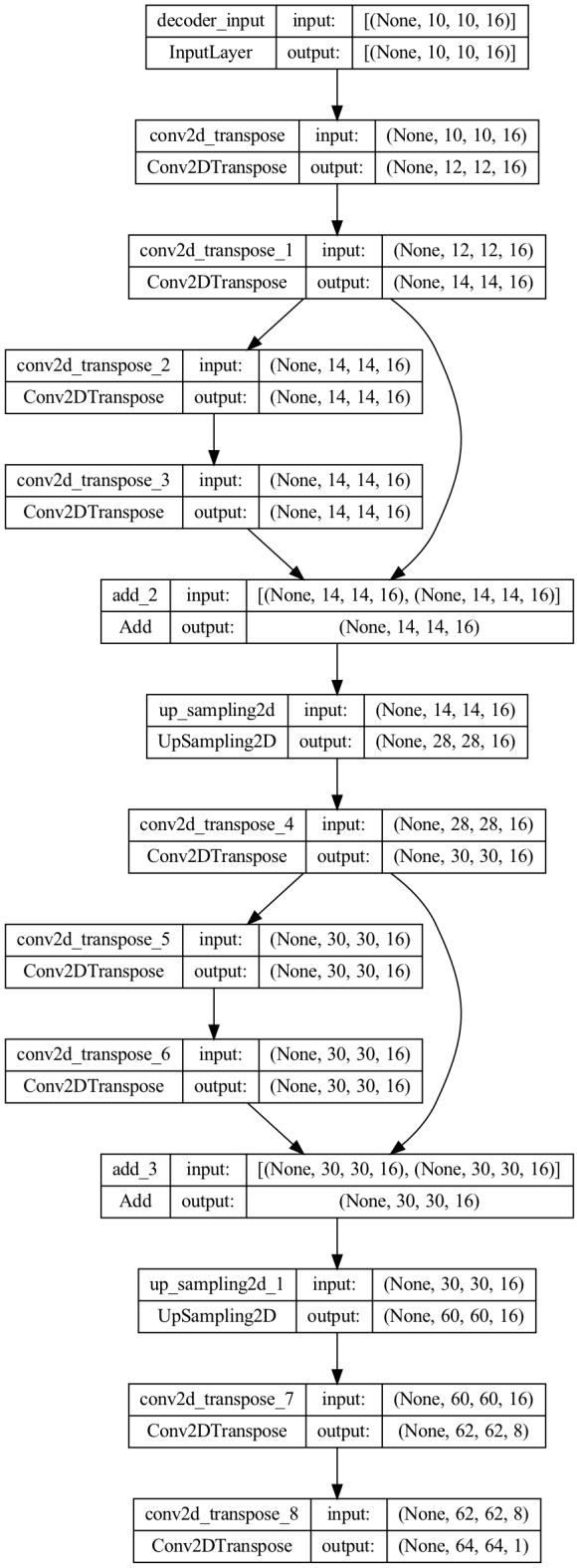
Schicht Name	Filters	Kernel Größe	Padding	pool_size
conv2d	8	3	no padding	none
conv2d_1	16	3	no padding	none
conv2d_2	16	3	same padding	none
max_polling2d	none	none	no padding	2
conv2d_3	16	3	same padding	none
conv2d_4	16	3	same padding	none
conv2d_5	16	3	same padding	none
conv2d_6	16	3	no padding	none
conv2d_7	16	3	no padding	none
max_polling2d_1	none	none	no padding	2
up_sampling2d	none	none	no padding	2
conv2d_transpose	16	3	no padding	none
conv2d_transpose_1	16	3	no padding	none
conv2d_transpose_2	16	3	same padding	none
conv2d_transpose_3	16	3	same padding	none
conv2d_transpose_4	16	3	same padding	none
conv2d_transpose_5	16	3	same padding	none
up_sampling2d_1	16	3	no padding	2
conv2d_transpose_6	8	3	no padding	none
conv2d_transpose_7	1	3	no padding	none

Tabelle 1: Die Einstellungen der Schichten des Encoders und Decoders.

Wie im Abschnitt 3.1 erklärt wurde, wurde ein Datensatz mit 64×64 Pixeln erstellt, um die arithmetischen Operationen mit mehreren Objekten zu untersuchen. Der Encoder kann Eingaben beliebiger Größe verarbeiten, jedoch kann sich die Komprimierungsrate ändern. Daher sollen in dieser Arbeit alle Experimente unter gleicher Kompressionsrate durchgeführt werden, weshalb ein Encoder mit Eingangsbildern von 64×64 Pixeln implementiert wird. Wenn Die Eingabe des Encoders 32×32 Pixel beträgt, ergibt sich eine Ausgabe von $5 \times 5 \times 16$, was einer Komprimierungsrate von 39,06% entspricht. Um dieselbe Komprimierungsrate beizubehalten, muss die Ausgabe bei einer Eingabe von 64×64 Pixeln $10 \times 10 \times 16$ betragen. Der Encoder mit Eingangsbildern von 64×64 Pixeln hat eine zusätzliche Convolutional-Schicht zwischen den residualen Blöcken. Die zusätzliche Schicht hat die folgenden Einstellungen: Filter 8, Kernel-Größe 3 und No-Padding. Die Einstellungen der anderen Schichten sind identisch mit denen des 32-Pixel-Modells. Abbildung 18a zeigt die Struktur des Encoders mit einer Eingabe von 64×64 , während Abbildung 18b die symmetrische Struktur des Decoders darstellt.



(a) Die Struktur des Encoders



(b) Die Struktur des Decoders

Abbildung 18: Struktur des Encoders und Decoders für ein Eingangs- und Ausgangsbild mit einer Größe von 64 $\times$ 64 Pixeln.

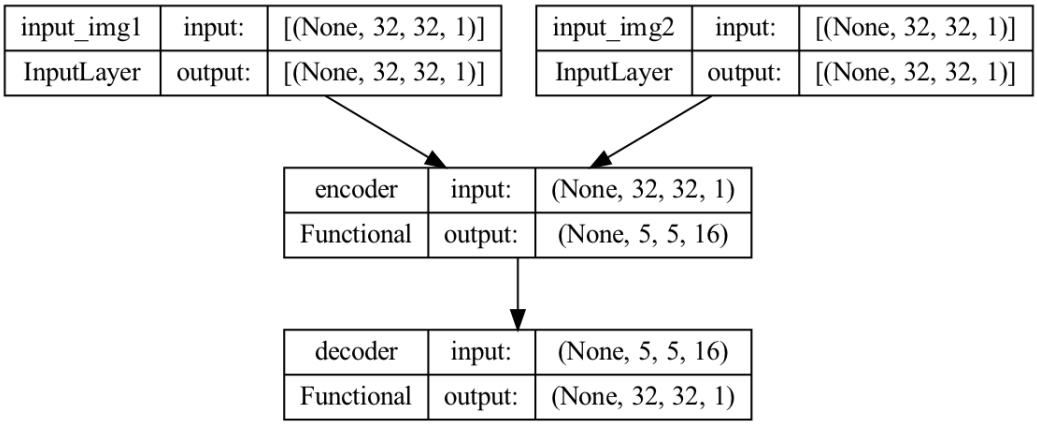
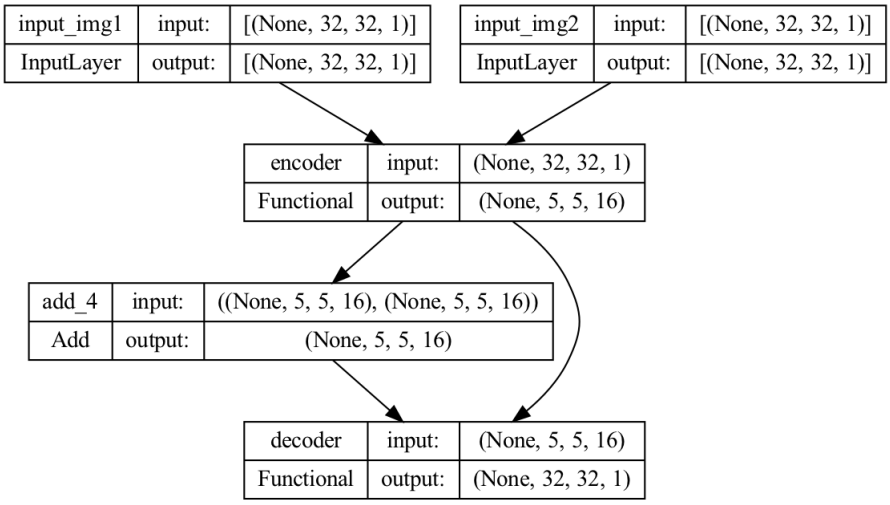


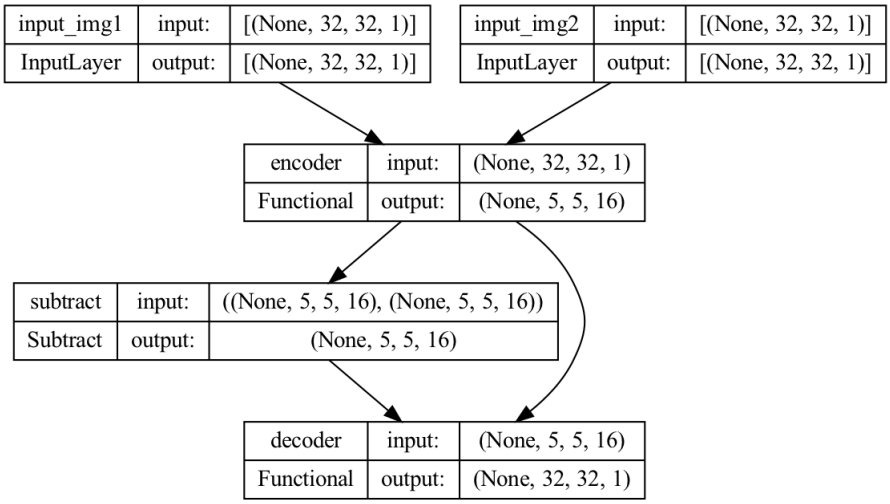
Abbildung 19: Die Struktur eines ResNet-Autoencoder ohne arithmetische Operationsschicht.

Der zuvor definierte Encoder und Decoder werden als Funktionen verwendet, um den Autoencoder aufzubauen. Dadurch hat der ResNet-Autoencoder eine klare Architektur, die aus Eingabe, Encoder und Decoder besteht. Die Ausgabe des Decoders ist die Ausgabe des ResNet-Autoencoders. Abbildung 19 zeigt die Architektur des ResNet-Autoencoders ohne die arithmetischen Operationsschichten. Das Modell ermöglicht den direkten Zugriff auf die kodierten Eingangsbilder für den Decoder. Die arithmetische Operationen im latenten Raum werden durch *'tf.keras.layers.add'* und *'tf.keras.layers.subtract'* realisiert. Abbildung 20a zeigt die Struktur des ResNet-Autoencoders mit einer Additionsschicht, während Abbildung 20b die Struktur des ResNet-Autoencoders mit einer Subtraktionsschicht darstellt. Die enkodierte Eingabedaten werden durch die Additions-oder Subtraktionsschicht berechnet.

In dieser Arbeit wurde versucht, das ResNet-Autoencoder mit zusätzlichen Kanteninformationen zu trainieren, um eine bessere Rekonstruktion der Objektkanten zu ermöglichen. Diese Kanteninformationen können mithilfe einer Kantendetektionsmethode wie dem Sobel-Filter aus den Originalbildern berechnet werden. Bei der Integration des Sobel-Filters musste die Struktur des Autoencoders modifiziert werden. Ein Sobel-Filter wird, wie in Abbildung 21 gezeigt, in das 64-Pixel-Modell integriert. Um die Kanteninformationen, die durch den Sobel-Filter generiert werden, mit den Originalbildern zu fusionieren, wird der Encoder nun in zwei Teile aufgeteilt. Die Einstellungen und die Reihenfolge der Schichten für Encoder und Decoder bleiben unverändert. Der Teil von *conv2d* bis zum ersten residualen Block wird als Encoder 1 definiert (vgl. Abbildung 18a). Der Teil von *conv2d_4* bis zu der Schicht *conv2d_8* wird als Encoder 2 bezeichnet (vgl. Abbildung 18a). Der Sobel-Filter befindet sich zwischen Eingabe-Schicht und Encoder 1. Die Eingabebilder und die Kanteninformationen aus dem Sobel-Filter werden in Encoder



(a) ResNet-Autoencoder mit einer Additionsschicht zwischen Encoder und Decoder



(b) ResNet-Autoencoder mit einer Subtraktionsschicht zwischen Encoder und Decoder

Abbildung 20: Die Struktur des ResNet-Autoencoders mit einer arithmetischen Operationsschicht.

1 kodiert. Zwischen Encoder 1 und Encoder 2 befindet sich eine Addition-Schicht, in der die kodierte Bilddaten und Kanteninformationen werden fusioniert. Die fusionierten Daten werden anschließend an Encoder 2 weitergeben. Die Operationsschicht und die Struktur des Decoders sind identisch mit dem vorherigen 64-Pixel-Modell.

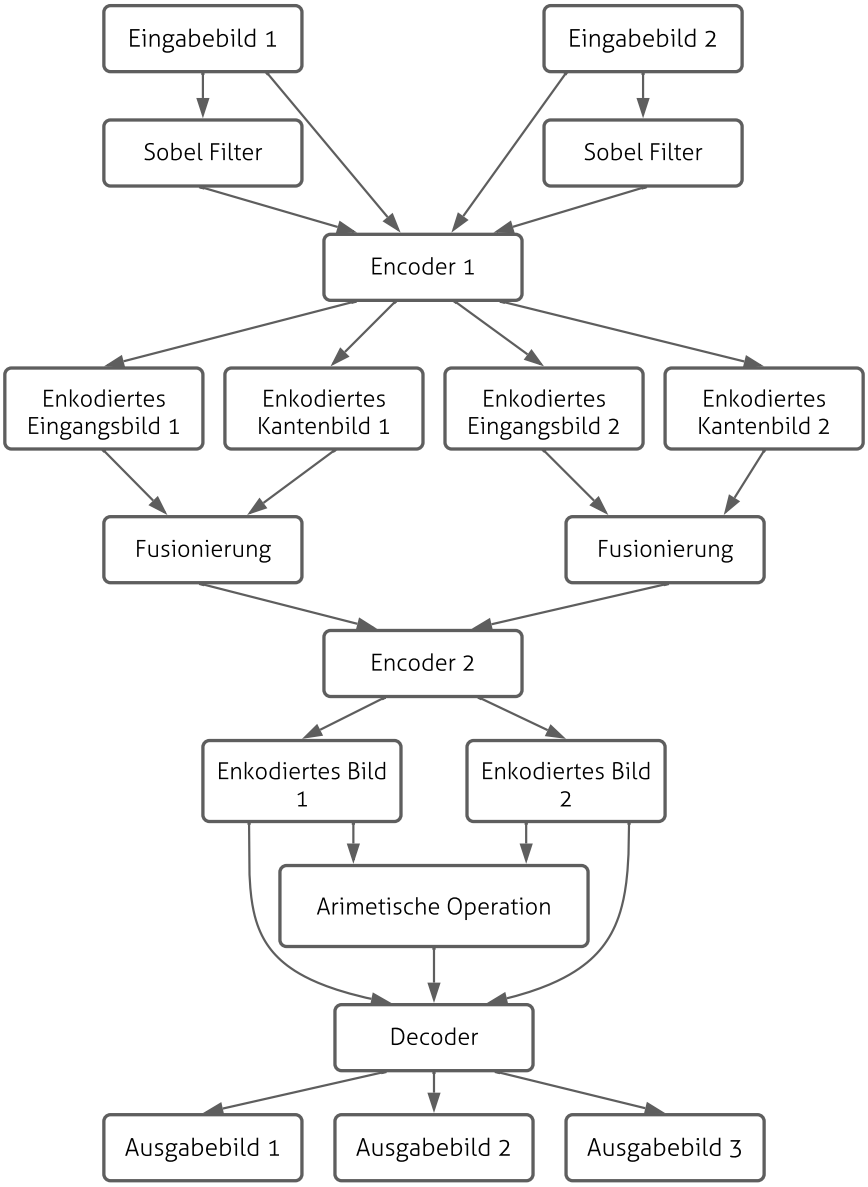


Abbildung 21: Die Struktur eines ResNet-Autoencoder mit Integration von Sobel-Filter.

4 Experimente und Ergebnisanalyse

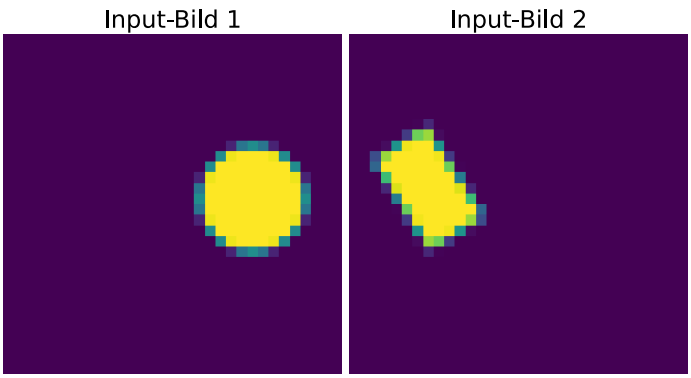
In diesem Kapitel werden sechs Experimente präsentiert und deren Ergebnisse analysiert.

4.1 Experiment 1: Vergleich von Convolutional-Autoencoder und ResNet-Autoencoder

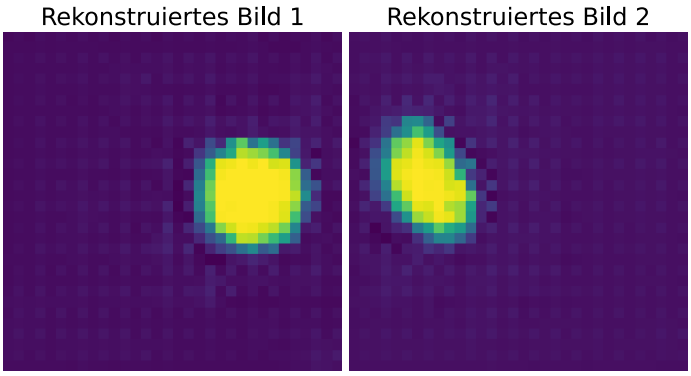
Zu Beginn wurde der ResNet-Autoencoder ohne arithmetische Schicht mit einem Convolutional-Autoencoder verglichen, um die Funktionalität des ResNet-Autoencoders zu prüfen. Zum Vergleich wurde der Convolutional-Autoencoder von Alexandru Dinu verwendet. Der originale Code des Convolutional-Autoencoders wurde in PyTorch geschrieben[10]. Die Dimension der Eingabe des Encoders beträgt 32×32 Pixel, die Ausgabe des Encoders beträgt $4 \times 4 \times 16$, was einer Komprimierungsrate von 25% entspricht. Um eine vergleichbare Komprimierungsrate zu erreichen, wurde der ResNet-Autoencoder mit einer Eingabe des Encoders von 32×32 Pixeln und einer Ausgabe des Encoders von $5 \times 5 \times 8$ entwickelt, was einer Komprimierungsrate von 19,5% entspricht. Die Filtergröße der Convolutional-Schichten im ResNet-Autoencoder wurde in diesem Experiment auf 8 festgelegt. Beide Autoencoder wurden mit einer Batch-Size von 100 und nur zehn Epochen trainiert. Die Verlustfunktion MSE wurde verwendet und der Optimizer war Adam.

Die rekonstruierten Bilder werden zum Vergleich verwendet. In Abbildung 22 werden die Ergebnisse beider Autoencoder graphisch dargestellt. Abbildung 22a zeigt die originale Bilder, von denen eines eine Ellipse und das andere ein Rechteck enthält. Abbildung 22b zeigt die rekonstruierten Bilder des Convolutional-Autoencoders. In den beiden rekonstruierten Bildern gibt es mehrere blaue Pixel als Rauschen im Hintergrund. Die Kanten der Objekte sind schlecht erkennbar. Abbildung 22c zeigt die rekonstruierten Bilder des ResNet-Autoencoders. Diese rekonstruierten Bilder haben kein Rauschen im Hintergrund und nur wenige Verzerrungen an den Kanten.

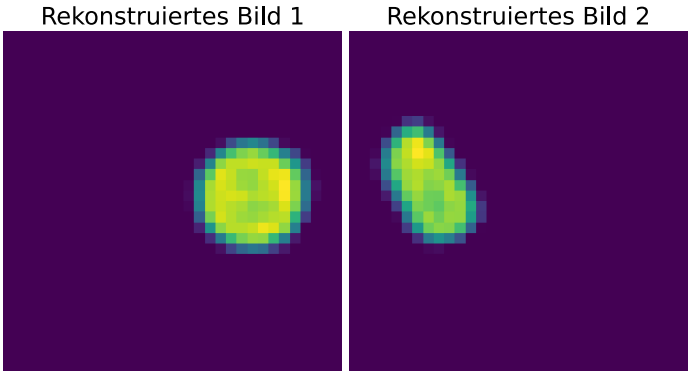
Die rekonstruierten Bilder des ResNet-Autoencoders haben einen klaren Hintergrund und bessere Formen als die des Convolutional-Autoencoders. Es konnte gezeigt werden, dass der ResNet-Autoencoder trotz einer stärkeren Komprimierungsrate bessere Ergebnisse lieferte. Diese Experimente bestätigen die Theorie im Abschnitt 2.3, dass der ResNet-Autoencoder eine höhere Leistungsfähigkeit als der Convolutional-Autoencoder aufweist.



(a) Originale Bilder



(b) Rekonstruierte Bilder eines Convolutional-Autoencoder



(c) Rekonstruierte Bilder eines ResNet-Autoencoder

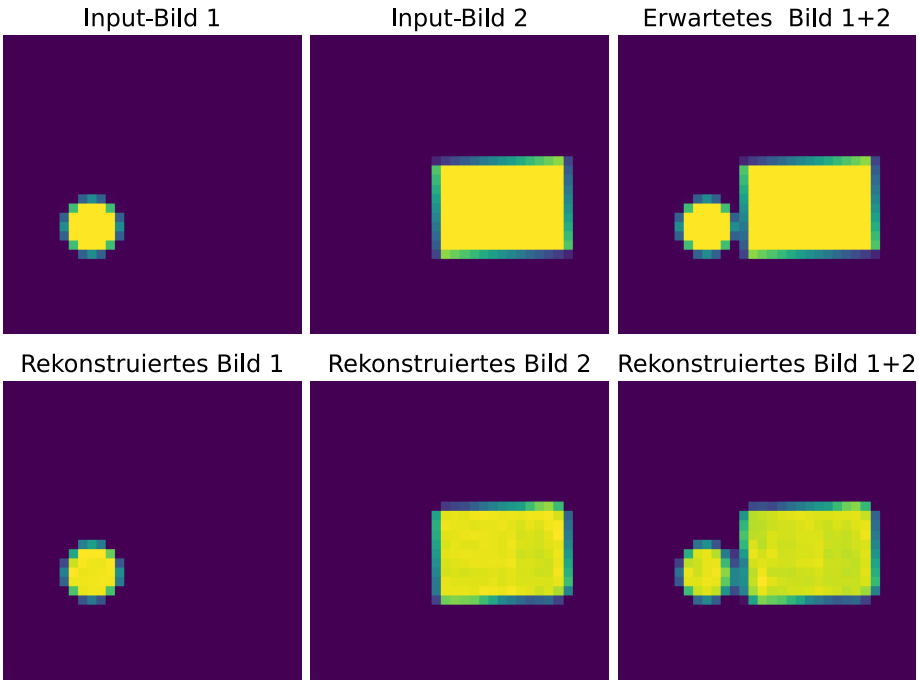
Abbildung 22: Vergleich der Ergebnisse von Convolutional- und ResNet-Autoencoder

4.2 Experiment 2: Arithmetische Operationen im latenten Raum

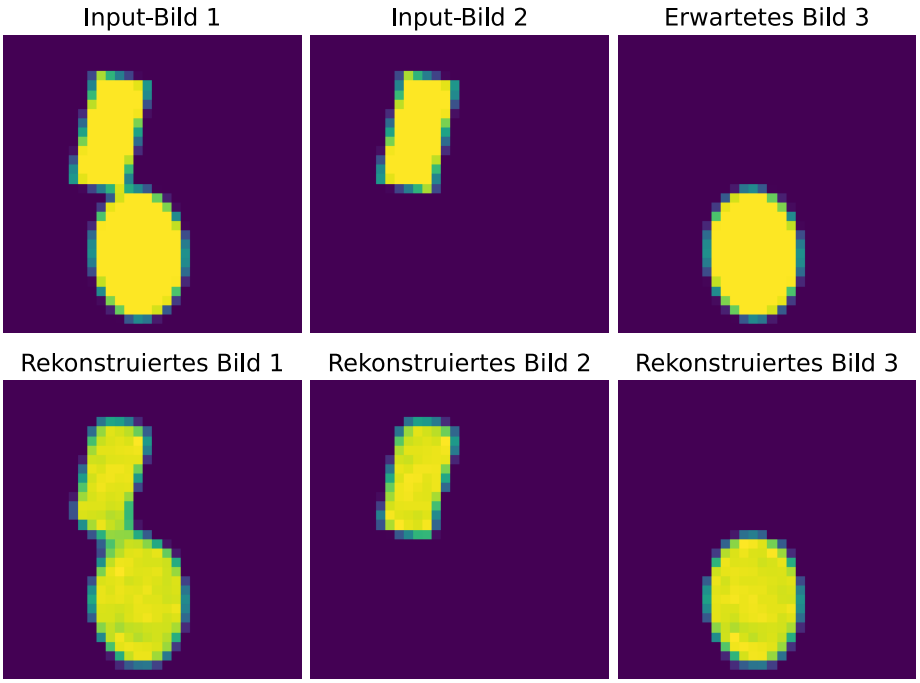
Weiterhin wurde nur der ResNet-Autoencoder betrachtet. Es wurden zwei Modelle aufgebaut, eines mit einer Additionsschicht und eines mit einer Subtraktionsschicht. Die Dimensionen der Eingabe und den latenten Vektoren von beiden Modell sind identisch, nämlich 32×32 und $5 \times 5 \times 16$. Beide Modelle wurden mit denselben Hyperparameter trainiert. Das Training der Modelle erfolgte über 30 Epochen mit einer Batch-Größe von 100. Der Trainingsdatensatz war identisch und hatte ein Überschneidungsverhältnis von 0,5.

In Abbildung 23 sind die Ergebnisse beider Modelle dargestellt. Abbildung 23a zeigt das Ergebnis des Addition-Modells, bei dem eine kleine Ellipse und ein großes Rechteck durch die Addition im latenten Raum überlagert werden. Abbildung 23b zeigt das Ergebnis des Subtraktions-Modells, bei dem ein Bild mit zwei Objekten auf ein Objekt reduziert wird. Ein Sample mit näheren Objekten und ein Sample mit leicht überlappenden Objekten werden rekonstruiert. Obwohl die rekonstruierten Bilder mehrere Rauschpixel (grüne Pixel) in den Objekten aufweisen, ist die Form aller Objekte gut erkennbar. In Abbildung 24 werden die Verluste beide Modelle währende der Epochen dargestellt. Die X-Achse zeigt die Anzahl der Epochen und die Y-Achse den MSE-Verlust. Die blaue Kurve ist das Training-Loss und die orange Kurve ist das Validation-Loss. Beim Addition-Modell sinkt der MSE-Wert in den ersten fünf Epochen schnell und danach sinkt er MSE-Wert langsam. Beim Subtraktion-Modell sinkt der MSE-Wert in den ersten 15 Epochen langsam und nähert sich nach 25 Epochen einem konstanten Wert an. Das Validation-Loss beider Modelle ähnelt dem Training-Loss. Es gibt nur minimale Schwankungen bei den Training- und Validation-Loss.

Die Ergebnisse zeigen, dass nach 30 Epochen Training beide Modelle in der Lage sind, die näheren und konturierteren Objekte im Bild gut zu erfassen. Das Addition- und Subtraktion-Modell können den Inhalt eines Bildes nach bestimmten Merkmalen erweitern und reduzieren. Es wurde gezeigt, dass die Überlagerung von zwei Bildern durch arithmetische Operationen im latenten Raum möglich ist. Interessanterweise kann das Subtraktion-Modell auch leicht überlappende Objekte richtig trennen.

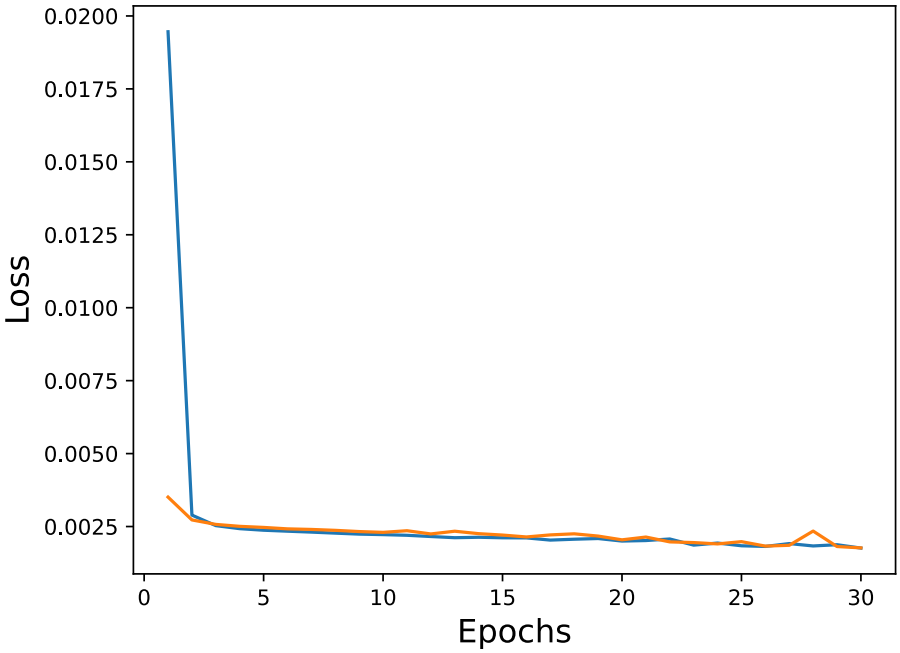


(a) Die Ergebnisse des Modells mit Additionsschicht

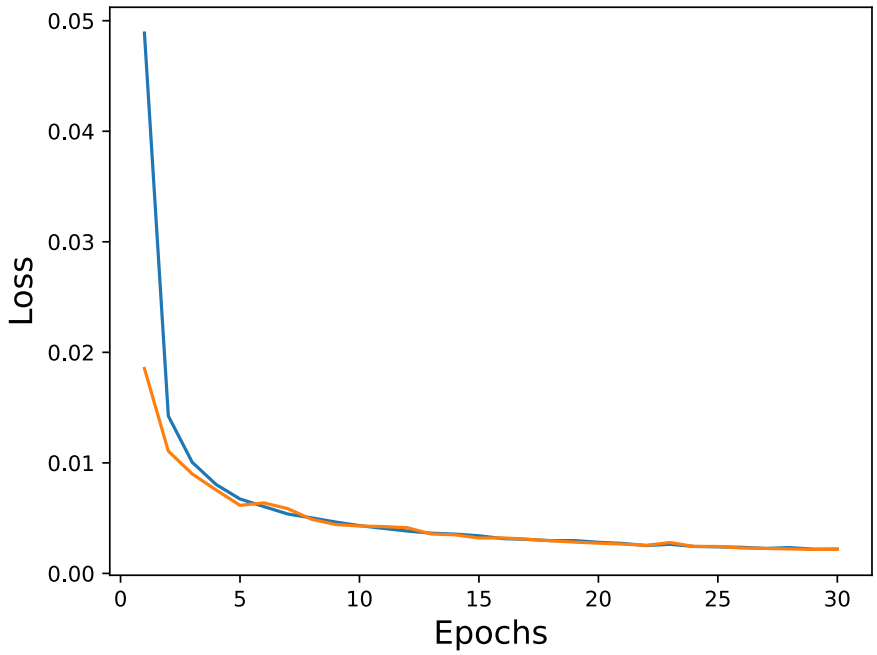


(b) Die Ergebnisse des Modells mit Subtraktionsschicht

Abbildung 23: Grafische Darstellung der Ergebnisse für den Modell mit Additionsschicht und Subtraktionsschicht



(a) Das Training- und Validation-Loss für Addition-Modell



(b) Das Training- und Validation-Loss für Subtraktion-Modell

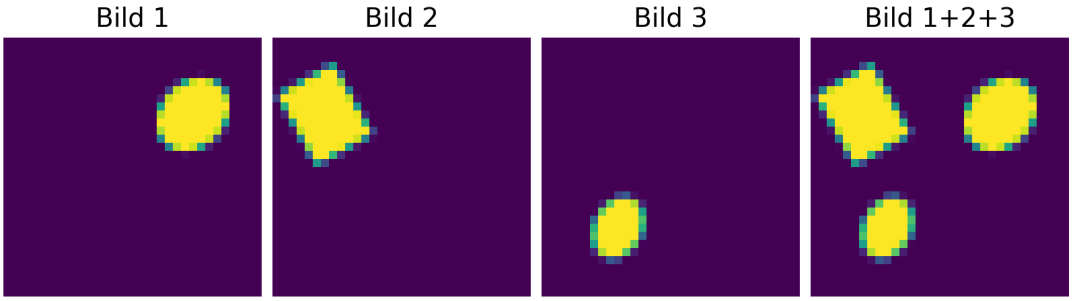
Abbildung 24: Grafische Darstellung von Training-und Validation-Loss für das Addition- und Subtraktion-Modell

4.3 Experiment 3: Aufbau des ResNet-Autoencoder mit trainierten Encoder und Decoder

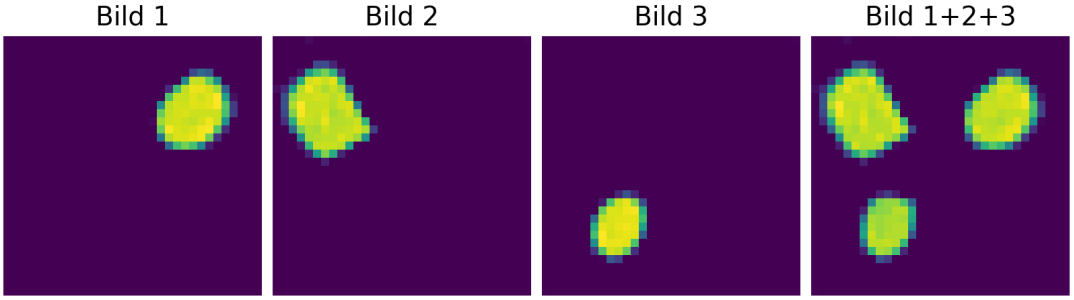
Der Lernalgorithmus für die Arithmetik wurde in diesem Experiment weiter untersucht. Zwei Modelle wurden mit einem trainierten Encoder und Decoder aufgebaut. Der verwendete Encoder und Decoder wurden im Experiment 2 trainiert und beide Modelle benötigen kein weiteres Training. Ein Modell ist ein Drei-Input-Modell mit drei Eingabe und vier Ausgabe. Das Modell versucht drei Objekte in einem Bild zu überlagern, um die Skalierbarkeit der Anzahl der Eingänge zu testen. Das andere Modell ist ein Subtraktion-Modell. Mit dem Subtraktion-Modell wurde es getestet, ob die arithmetische Operationen skalierbar sind.

In Abbildung 25 wird die Vorhersage des Drei-Input-Modells dargestellt. Die Bilder in Abbildung 25a zeigen die Originalbilder, während die Bilder in Abbildung 25b die rekonstruierten Bilder darstellen. Aus der Grafik geht hervor, dass drei Objekte korrekt in einem Bild überlagert wurden. Allerdings gibt es Verzerrungen in den Objekten, da viele grüne Pixel vorhanden sind, was auf einen höheren MSE-Verlust hinweist. Die Vorhersagen des Subtraktion-Modells werden in Abbildung 26 dargestellt. Die originalen Bilder und die rekonstruierten Bilder werden in ersten und zweiten Reihe dargestellt. Das Modell ist in der Lage, zwei nahe Objekte zu erkennen und durch Subtraktion richtig zu trennen. Allerdings werden die Kanten der nahen Objekte nicht gut erkannt was zu Verzerrungen in den überlappenden Bereichen führt. Ein hoher MSE-Verlust wird durch die vielen grünen Pixel im Objekt angezeigt.

Anhand dieser Ergebnisse wird deutlich, dass der Encoder und Decoder, die mit einer Additionsschicht trainiert wurden, ohne weiteres Training mit mehreren Additionsschichten oder Subtraktionsschicht funktionieren können. Das bedeutet, dass sowohl die Anzahl der Eingänge als auch die arithmetische Operationen skalierbar sind.



(a) Originale Bilder



(b) Rekonstruierte Bilder

Abbildung 25: Die Vorhersagen des trainierten Addition-Modells mit drei Eingaben und vier Ausgaben

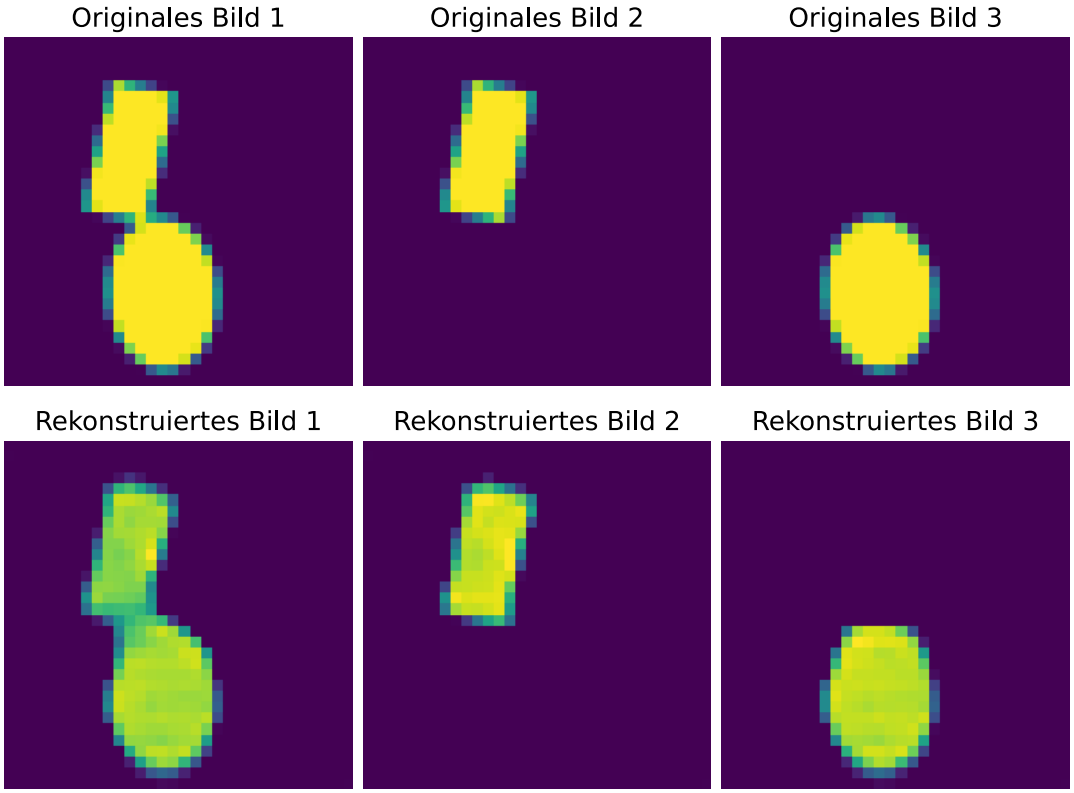


Abbildung 26: Die Vorhersagen des trainierten Subtraktion-Modells.

4.4 Experiment 4: Mehrere Objekte in einem Bild zu überlagern

In diesem Experiment wurden nacheinander mehrere Objekte in ein Bild überlagert, um die Grenzen des Addition-Modells zu untersuchen. Es wurde ein Addition-Modell mit Eingaben von 64×64 Pixeln erstellt. Dieses Modell hat zwei Eingaben und drei Ausgaben. Zunächst wurde das Modell mit einer Batch-Size von 80 über 40 Epochen trainiert und anschließend gespeichert. Um sicherzustellen, dass das gespeicherte Modell mehrere Bilder addieren kann, wurde ein Loop-Programm geschrieben. Das Ablaufdiagramm des Loop-Programms wird in Abbildung 27 dargestellt. Das Diagramm zeigt den Ablauf des Loop-Programms, welches verwendet wurde, um das gespeicherte Addition-Modell auf mehrere Bilder anzuwenden. In der Schleife wurde das überlagerte Ausgangsbild des Modells als Eingangsbild verwendet. Ein Datensatz mit neun Bildern wurde für die Vorhersage verwendet. Acht dieser Bilder enthalten jeweils ein Objekt, das neunte Bild zeigt die Überlagerung aller acht Bilder. Es gibt keine überlappenden Objekte in diesem Datensatz.

Das Originalbild in Abbildung 28a enthält alle acht Objekte. In Abbildung 28b, 28c und 28d sind die Ergebnisse der Addition von zwei, fünf und acht Objekten zu sehen. In Abbildung 28b sind die kanten den Zwei Rechtecke noch gut zu erkennen. Nach der Addition von weiteren drei Objekten sind die Kanten der ersten beiden Rechtecke nicht mehr gerade. Die Formen der neuen überlagerten Rechtecke und Ellipse sind erkennbar. Nach der Addition weiterer drei Objekte wird es immer schwieriger, die Form der ersten beiden Rechtecke zu erkennen. Alle Objekte in Abbildung 28d weisen starke Verzerrungen auf. Eine kleine Ellipse wird als Rechteck rekonstruiert.

Aus den Ergebnissen ist zu sehen, dass mit zunehmender Anzahl an hinzugefügten Objekten immer mehr Verzerrungen auftreten. Obwohl bei der Addition von acht Objekten die Positionen der Objekte korrekt sind, sind die Formen der Objekte stark verzerrt. Die Analyse der Ergebnisse zeigt, dass ein trainiertes Addition-Modell in der Lage ist, mehrere Bilder nacheinander zu überlagern. Um eine bessere Leistung zu erzielen, sollte die Anzahl der Bearbeitungen pro Bild auf weniger als vier begrenzt werden.

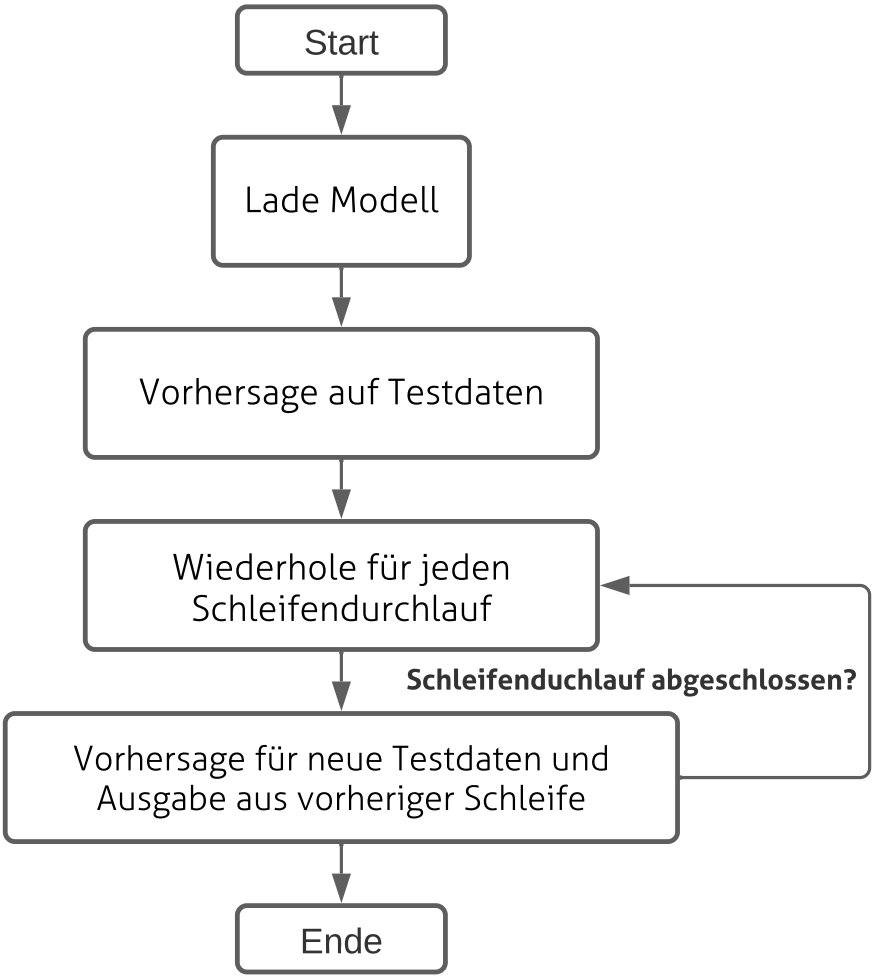
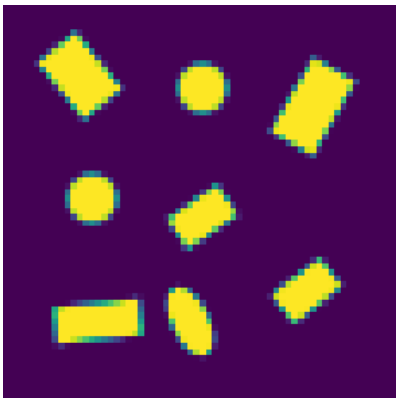
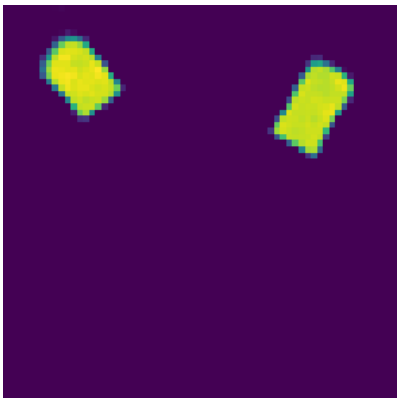


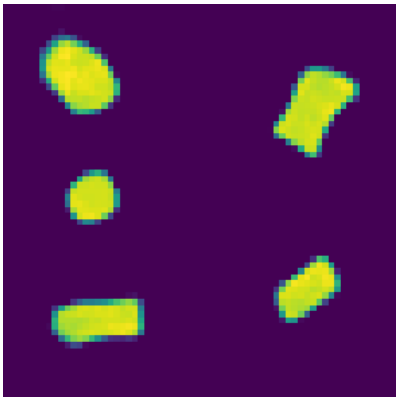
Abbildung 27: Ablauf des Experiments



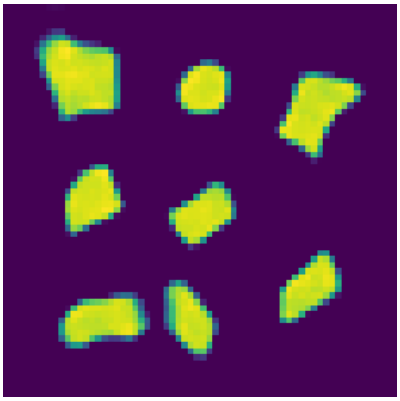
(a) Das original überlagerte Bild



(b) Rekonstruriertes Bild mit zwei überlagerten Objekte



(c) Rekonstruriertes Bild mit Fünf überlagerten Objekte



(d) Rekonstruriertes Bild mit Acht überlagerten Objekte

Abbildung 28: Ergebnisse des Loop-Programms. Acht Objekte nacheinander in einem Bild überlagert.

4.5 Experiment 5: Tuning von Learning Rate und Batch Size

Im diesem Abschnitt wurde der 64 $\times$ 64 Pixel ResNet-Autoencoder mit verschiedenen Learning-Rates und Batch-Sizes trainiert, um die beste Kombination von Learning-Rate und Batch-Size zu finden, damit das Modell gut trainiert werden kann. Die Learning-Rates wurden auf die Werte 0,001, 0,003 und 0,005 eingestellt. Die Batch-Sizes wurden auf die Werte 20, 80 und 140 eingestellt. Insgesamt wurde das Modell mit neun Einstellungen für Learning-Rate und Batch-Size separat trainiert. Alle Trainings wurden über 40 Epochen durchgeführt.

In Tabelle 2 sind die Einstellungen des Trainings und die zugehörigen MSE-Werte für jedes Training dargestellt. Die Tabelle zeigt, dass das Training 2 den niedrigsten MSE-Wert hat.

Training	Batch Size	Learning Rate	MSE	SSIM
1	20	0,001	$\mathbf{2{,}216 \times 10^{-4}}$	**0,9965**
2	20	0,003	$\mathbf{2{,}129 \times 10^{-4}}$	**0,9967**
3	20	0,005	$\mathbf{2{,}533 \times 10^{-4}}$	**0,9962**
4	80	0,001	$\mathbf{3{,}973 \times 10^{-4}}$	**0,9943**
5	80	0,003	$\mathbf{3{,}059 \times 10^{-4}}$	**0,9955**
6	80	0,005	$\mathbf{3{,}823 \times 10^{-4}}$	**0,9951**
7	140	0,001	$\mathbf{5{,}561 \times 10^{-4}}$	**0,9924**
8	140	0,003	$\mathbf{5{,}662 \times 10^{-4}}$	**0,9917**
9	140	0,005	$\mathbf{5{,}067 \times 10^{-4}}$	**0,9928**

Tabelle 2: Darstellung der Einstellungen für alle neun Trainings und ihrer jeweiligen MSE-und SSIM-Werte.

Das Training-Loss des Trainings 1, 4 und 7 wird in Abbildung 29 dargestellt. Alle drei Training haben dieselbe Learning-Rate von 0,001. Die grüne Kurve, orange Kurve und blaue Kurve im Diagramm zeigen jeweils das Training-Loss für eine Batch-Size von 140, 80 und 20. Die blaue Kurve (Batch-Size 20) hat von Anfang an das kleinste Training-Loss und das Training-Loss konvergiert am schnellsten in allen drei Training. Während des Trainings ist das Loss der blauen Kurve immer am niedrigsten. Abbildung 30 zeigt das Training-Loss der Einstellungen 1, 2 und 3, die alle eine Batch-Size von 20 haben. Die grüne Kurve, orange Kurve und blaue Kurve im Diagramm zeigen jeweils das

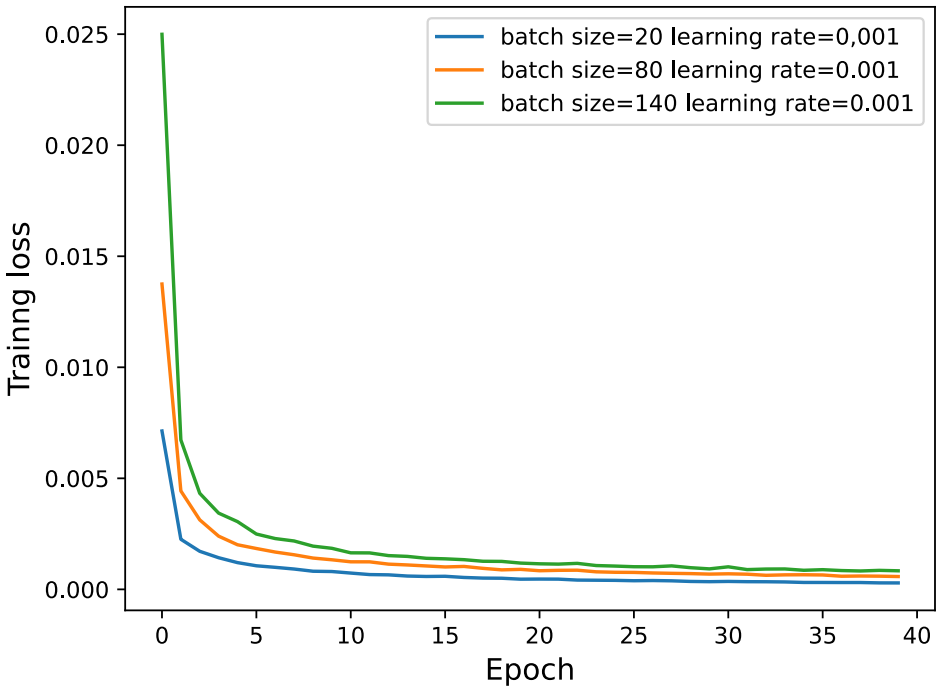


Abbildung 29: Vergleich des Trainingsverlusts für Batch-Size 20, 80 und 140 bei derselben Learning-Rate von 0,001.

Training-Loss für Learning-Rates von 0,005, 0,003 und 0,001. Eine zusätzliche Zoom-In-Grafik zeigt die Loss-Kurve zwischen Epoch 5 und Epoch 10. Das Training-Loss für die orange Kurve (Learning-Rate von 0,003) konvergiert in den ersten zehn Epochen am schnellsten. Nach Epoch 15 sind die Training-Losses für die Learning-Rates von 0,003 und 0,001 fast gleich. Die grüne Kurve (Learning-Rate von 0,005) hat im gesamten Trainingsverlauf das höchste Training-Loss. Das Training-Loss aller neun Einstellungen werden in Abbildung 31 dargestellt. Eine Zoom-In-Grafik zeigt die Loss-Kurve zwischen Epoch 1 und Epoch 6. Die Abbildung 31zeigt, dass die orange Kurve (Batch-Size ist 20, Learning-Rate ist 0,003) am schnellsten konvergiert und immer den niedrigsten Verlust aufweist. Die Ergebnisse von Training 2 (mit der niedrigsten MSE) und von Training 8 (mit der höchsten MSE) werden in Abbildung 32 anschaulich dargestellt. Hier wird ein Sample mit überlappenden Objekten rekonstruiert. Die rekonstruierten Bilder des Trainings 8 (siehe Abbildung 32b) enthalten mehr grüne Pixel als die des Trainings 2 (siehe Abbildung 32c). In der Farbkarte *Viridis* repräsentiert grüne Pixel eine Abweichung vom Wert 1. Je mehr grüne Pixel ein rekonstruiertes Bild hat, desto größer ist die Abweichung vom Originalbild. Die grafischen Ergebnisse zeigen, dass der MSE-Wert des Trainings 8 höher ist als der des Trainings 2. Training 2 ist besser in der Lage, die Konturlinien im überlappenden Bereich zu rekonstruieren.

In diesem Versuch wurde gezeigt, dass die Hyperparameter Learning-Rate und Batch-Size die Konvergenzgeschwindigkeit des Trainingsverlusts und des Trainingsergebnisses beeinflussen können. Im Allgemeinen konvergiert die Verlustkurve schneller bei einer geringen Learning-Rate und einer geringen Batch-Size. Der SSIM zeigt ähnliche Ergebnisse, bei denen eine Learning-Rate von 0,003 zu einem höheren SSIM-Wert hat. Jedoch garantieren eine geringe Learning-Rate nicht zwangsläufig einen geringen MSE-Wert. Zum Beispiel ist der MSE-Wert für Training 9 mit einer Learning-Rate von 0,005 kleiner als der für Training 7 mit einer Learning-Rate von 0,001. Die beste Kombination von Batch-Size und Learning-Rate sollte je nach dem Problem sorgfältig untersucht werden.

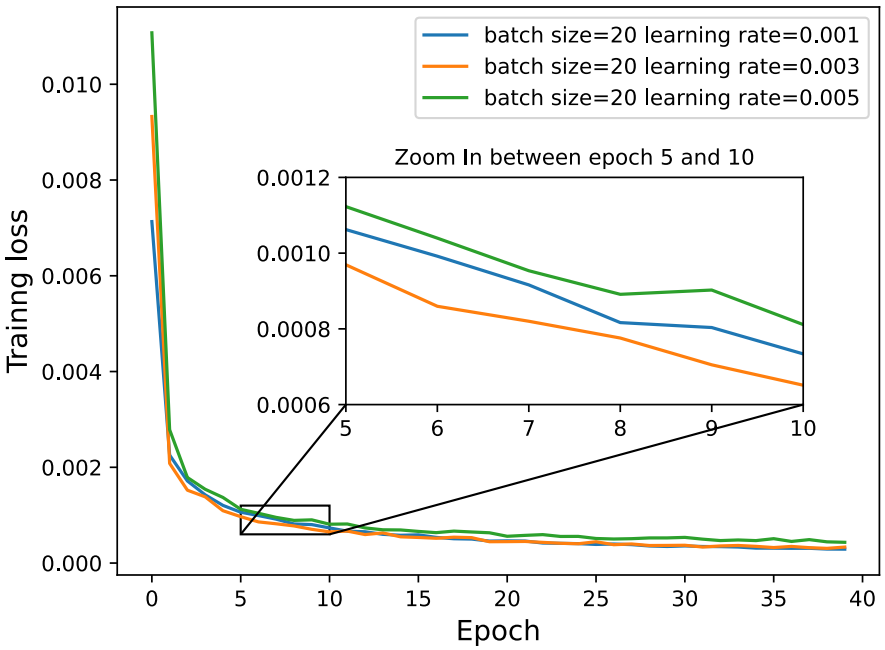


Abbildung 30: Vergleich des Trainingsverlusts für Learning-Rate 0,001, 0,003 und 0,005 bei derselben Batch-Size von 0,001.

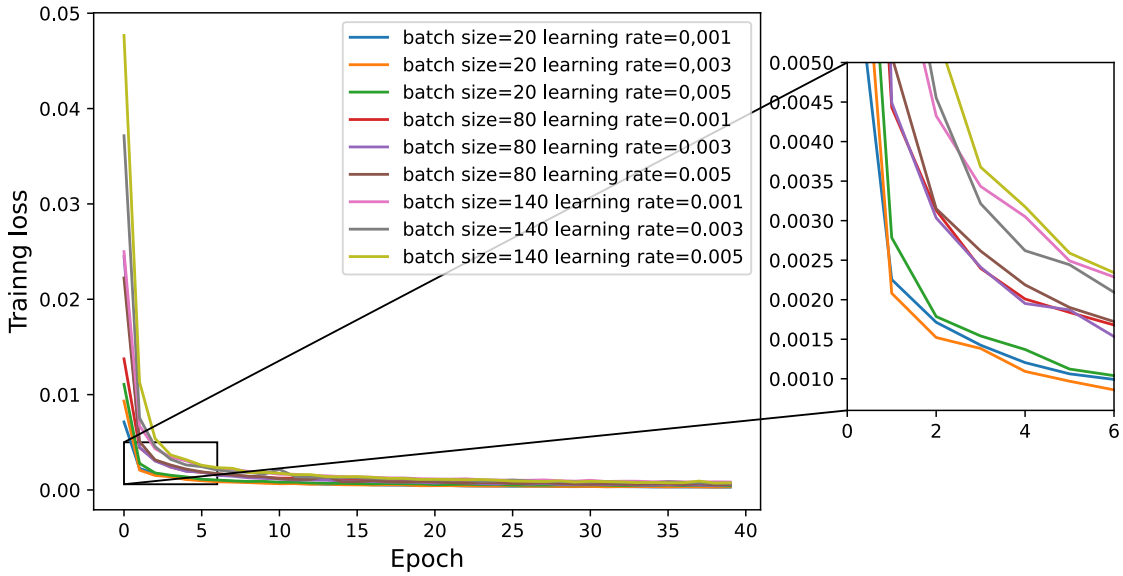
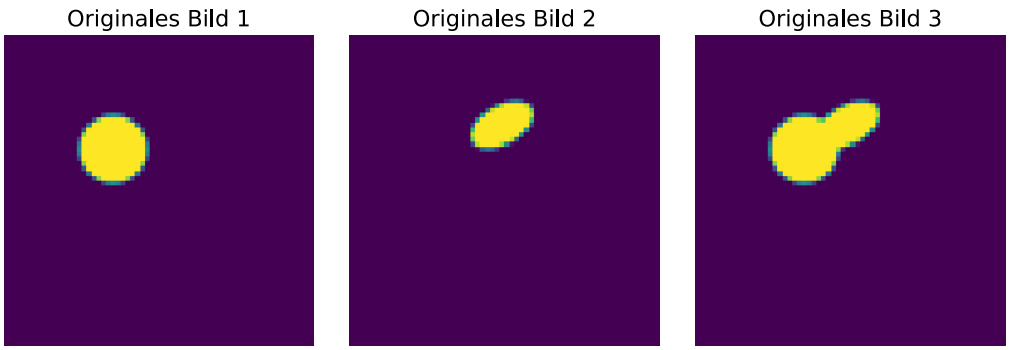


Abbildung 31: Trainingsverluste für alle neun Trainings.



(a) Originale Bilder

(b) Rekonstruierte Bilder aus Training 8

(c) Rekonstruierte Bilder aus Training 2

Abbildung 32: Darstellung der Ergebnisse von Training 8 und Training 2. Ein Beispiel-Sample mit überlappenden Objekten wird hier rekonstruiert.

4.6 Experiment 6: Integration des Sobel-Filters

Die bisherigen Experimente haben gezeigt, dass der ResNet-Autoencoder nach 30 bis 40 Epochen Training die Fähigkeit hat, die Kanten einzelner und nicht überlappender Objekte gut zu rekonstruieren. Allerdings werden bei wenigen Trainingsepochen (siehe Abbildung 22c) oder bei überlappenden Objekten (siehe Abbildung 24b) die Kanten nicht gut rekonstruiert. In diesem Experiment wurden zusätzliche Kanteninformationen aus dem Sobel-Filter in das Netz eingespeist, um dem ResNet-Autoencoder eine bessere Lernfähigkeit für die Kanten im überlappenden Bereich zu ermöglichen. Es wurde erwartet, dass das Netz mit Hilfe der Kanteninformationen auch bei wenigen Epochen Training gute Ergebnisse liefern kann. Daher wurden zwei Modelle trainiert. Ein Modell wurde wie Training 2 im letzten Versuch aufgebaut und das andere Modell wurde mit der Integration des Sobel-Filters erstellt. Das Modell mit Sobel-Filter hat vier Eingaben, wobei zwei davon die Kantenbilder aus dem Sobel-Filter sind. Die originale Bilder und die Kantenbilder werden im Encoder fusioniert und weiterverarbeitet. Beide Modelle wurden mit einer Learning-Rate von 0,003 und einer Batch-Size von 20 über 15 Epochen trainiert. Nach dem Training wurden die Vorhersagen beider Modelle anhand von 1000 Test-Samples ausgegeben. Anschließend wurden die SSIM-Werte und die MSE-Werte beider Modelle berechnet und analysiert.

In Tabelle 3 zeigt die MSE-Werte und der durchschnittliche SSIM für beide Modelle. Das Modell ohne Sobel-Filter hat einen MSE-Wert von $3{,}432 \times 10^{-4}$ und einen SSIM von 0,9952, während das Modell mit Sobel-Filter einen niedrigen MSE-Wert und höheren SSIM von $3{,}225 \times 10^{-4}$ und 0,9956. Die rekonstruiert Bilder von beiden Modell werden in Abbildung 33 dargestellt. Die grafische Darstellung der Ergebnisse zeigt auch, dass das Modell mit Sobel-Filter (rechtes Bild) die Kanten im überlappenden Bereich besser rekonstruiert. Um die strukturelle Ähnlichkeit des Ergebnisse zu analysieren, wird ein Histogramm in Abbildung 34 die SSIM-Werte beider Modelle dargestellt. Die SSIM-Werte auf der X-Achse sind in folgende fünf Gruppen eingeteilt: kleiner als 0,995, 0,995 bis 0,996, 0,996 bis 0,997, 0,997 bis 0,998 und größer als 0,998. Die Y-Achse zeigt die Anzahl der jeweiligen Gruppen. Das Modell ohne Sobel-Filter wird in orange dargestellt

Modell	Epoch	MSE	Durchschnittlicher SSIM
Ohne Sobel-Filter	15	$\mathbf{3{,}432 \times 10^{-4}}$	**0,9952**
Mit Sobel-Filter	15	$\mathbf{3{,}225 \times 10^{-4}}$	**0,9956**

Tabelle 3: Der MSE und SSIM für das Modell mit und ohne Sobel-Filter.

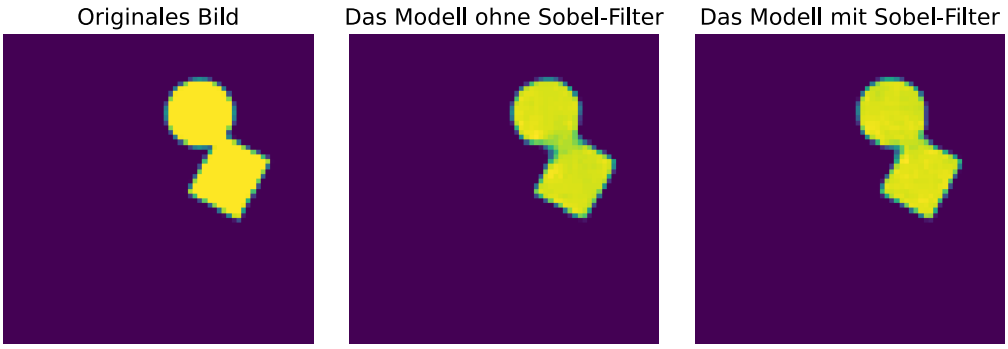


Abbildung 33: Grafische Darstellung der Ergebnisse des Modells ohne und mit Sobel-Filter.

und das Modell mit Sobel-Filter wird in blau dargestellt. Das Modell mit Sobel-Filter erzielte insgesamt 388 Vorhersagen mit einem SSIM-Wert größer als 0,997, während das Modell ohne Sobel-Filter insgesamt 298 Vorhersagen mit einem SSIM-Wert größer als 0,997 aufwies. Dieses Ergebnis zeigt, dass der Sobel-Filter dem Netzwerk hilft, mehrere gute Bilder zu rekonstruieren. Die Verteilung der SSIM-Werte des Modells mit Sobel-Filter ist kompakter als die Verteilung der SSIM-Werte des Modells ohne Sobel-Filter.

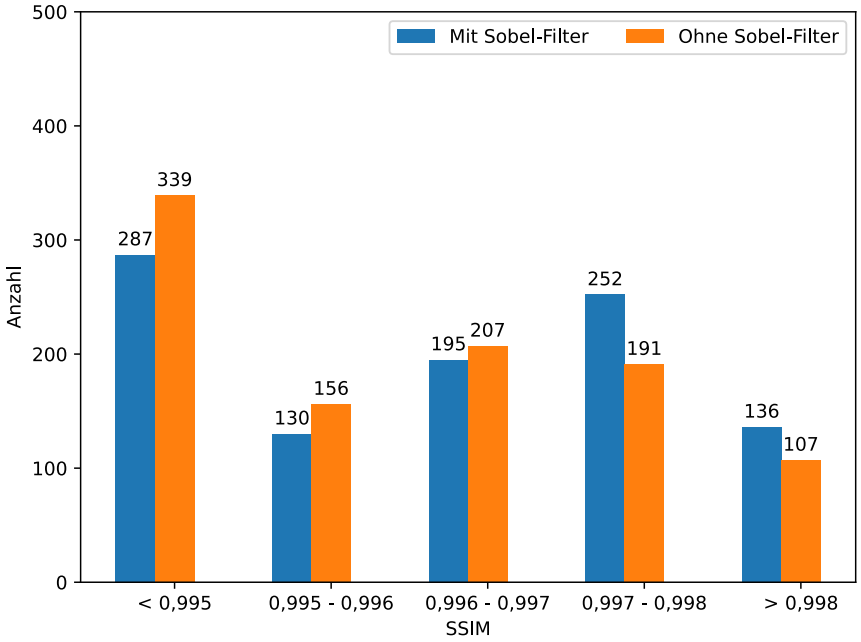


Abbildung 34: Darstellung der SSIM-Werte in verschiedenen Gruppen.

Aus den Trainingsergebnissen geht hervor, dass die Kanteninformationen aus dem Sobel-Filter eine Verbesserungen für den ResNet-Autoencoder gebracht hat. Diese zusätzliche Kanteninformationen helfen das Autoencoder die Kanten insbesondere bei überlappenden Objekte besser zu rekonstruieren. Die durchschnittlichen SSIM-Werte beider Modelle sind ähnlich. Allerdings weist das Modell mit Sobel-Filter mehr rekonstruierte Bilder im höheren SSIM-Wertebereich auf. Die Anzahl der SSIM-Werte, die größer als 0,997 sind, ist beim Modell mit Sobel-Filter um 90 höher als beim Modell ohne Sobel-Filter. Dies deutet darauf hin, dass die Integration zusätzlicher Informationen die Leistung des ResNet-Autoencoders verbessern kann. Es wäre sinnvoll, weitere Untersuchungen zur Integration des Sobel-Filters durchzuführen. Aufgrund der Berechnung der Kanteninformationen erfordert das Training beim Modell mit Sobel-Filter mehr Zeit.

5 Fazit

Zum Abschluss dieser Forschungsarbeit werden die Inhalte und Erkenntnisse zusammengefasst und bewertet. Darüber hinaus wird ein Blick auf möglich zukünftige Verbesserungen und Erweiterungen des ResNet-Autoencoders gegeben.

5.1 Zusammenfassung

Im Rahmen dieser Arbeit wurde ein Autoencoder mit Eingabedaten von 32×32 Pixeln sowie eine Variante mit Eingabedaten von 64×64 Pixeln entwickelt. Ziel war es, mit Hilfe des entwickelten Autoencoders zu untersuchen, ob es im latenten Raum möglich ist, arithmetische Operationen mit dem Bildinhalt durchzuführen, und wo die Grenze für mehrfache Anwendungen von Arithmetik liegt. Für das Training wurden mehre Datensätze erstellt, die Ellipsen und Rechtecke als Objekte enthalten. Die Datensätze wurden entsprechend der Größe des Netzes in 32×32 und 64×64 aufgeteilt. Als Maßnahme zum Vergleich den Ergebnisse wurden verschiedene Metriken eingesetzt. Zwei numerische Metriken sind MSE-und SSIM-Wert. Dabei repräsentiert der Wertebereich von 0 bis 1 verschiedene Farben. Der MSE-Wert kann durch den Vergleich der Pixelfarben anschaulich bewertet werden. Es gingen die erste Anforderung an dem Autoencoder hervor, welche Architektur des Autoencoders die Bilder möglichst gut rekonstruieren kann. Ein Convolutional-Autoencoder und ein ResNet-Autoencoder würden hierbei um die Genauigkeit der Rekonstruktion verglichen. Aus Experiment lässt sich feststellen, Ein ResNet-Autoencoder die Unterschiede zwischen die Objekte und das Hintergrund im Bild gut erkennen und die Bilder in höherer Genauigkeit rekonstruieren kann. Die Addition- und Subtraktionsschicht, die zwischen dem Encoder und dem Decoder implementiert wurden, sind in dem Sinn keine Komponenten des ResNet-Autoencoders. Sie wurden eingefügt, um arithmetische Operationen im latenten Raum zu testen. Die Experimente zeigen, dass eine Überlagerung oder Reduzierung des Bildinhalts durch Rechenoperationen auf den latenten Vektoren möglich ist. Es wurden verschiedene Experimente durchgeführt, um die Lernfähigkeit des ResNet-Autoencoders zu testen. Ein Experiment ergab, dass das trainierte Modell in der Lage ist, den Bildinhalt sowohl zu überlagern als auch zu reduzieren. Ein Erkenntnis aus den Experimenten ist, dass beim Training des ResNet-Autoencoders nur der Encoder und der Decoder als Teilkomponenten trainiert werden. Die implementierte arithmetische Schicht wird nicht beim Training einbezogen. Die Anzahl der Eingänge und die arithmetische Operationen eines trainierten Modells sind skalierbar. Die Grenze für mehrfache Additionen wurde

durch ein zuvor trainiertes Modell getestet. Bei mehrfacher Addition von Objekten tritt im rekonstruierten Bild eine Verzerrung des Objekts auf. Je mehr einzelne Objekte addiert werden, desto stärker tritt die Verzerrung auf. Anschließend würde der Sobel-Filter im ResNet-Autoencoder implementiert. Die aus dem Sobel-Filter berechneten Kanteninformationen verbessern die Fähigkeit des Netzes, die Kanten überlappender Objekte zu rekonstruieren.

Es hat sich als schwierig erwiesen, einen gut verteilten Datensatz von 32×32 Pixeln mit mehr als zwei Objekten im Bild zu erstellen. Da für drei große Objekte nicht genügend Platz vorhanden ist, ergibt sich eine nicht-normale Verteilung der Objektgrößen. Es wird ein größeres Bild für den Datensatz benötigt, was wiederum ein größeres Eingangsbild für den Autoencoder erfordert. Das Ändern der Eingabe- und Ausgabedaten von 32×32 auf 64×64 Pixel ändert die Kompressionsrate, daher muss die Architektur des ResNet-Autoencoders leicht angepasst werden, beispielsweise durch Hinzufügen von zusätzlichen Convolutional-Schichten. Basierend auf diesen Erkenntnissen wird im folgenden Abschnitt ein Ausblick auf mögliche Erweiterungen des ResNet-Autoencoders gegeben.

5.2 Ausblick

In diesem Abschnitt soll ein Ausblick auf mögliche weitere Forschung im Bereich des arithmetischen Bildinhalts gegeben werden. Bisher wurden alle Modelle mit Datensätzen trainiert und getestet, die nur Ellipsen oder Rechtecke enthalten. Ein weiteres Experiment könnte durchgeführt werden, bei dem der ResNet-Autoencoder mit einem Trainingsdatensatz, der nur Ellipsen und Rechtecke enthält, trainiert und dann auf einem Testdatensatz, der zusätzlich Dreiecke enthält, getestet wird. Dadurch könnte getestet werden, ob der ResNet-Autoencoder in der Lage ist, Bildinhalte mit unbekannten Objekten zu überlagern oder zu reduzieren. Ein weiteres interessantes Experiment wäre die Verwendung von Sobel-Filtern bei Bildern mit verrauschtem Hintergrund. Dieses Experiment könnte zeigen, wie gut der Sobel-Filter den ResNet-Autoencoder verbessert, um die Form des Objekts in Bildern mit schlechter Qualität zu rekonstruieren. Ein weiterer Schritt der Arbeit könnte die Möglichkeit untersuchen, ob die extrahierten und berechneten Merkmale des Bildinhalts auch in anderen Anwendungen, wie beispielsweise der Klassifizierung, eingesetzt werden können.

Literatur

[1] Mohamed Abdel-Basset, Nour Moustafa und Hossam Hawash. „Autoencoder Networks“. In: *Deep Learning Approaches for Security Threats in IoT Environments*. 2023, S. 249–269. DOI: 10.1002/9781119884170.ch11.

[2] Arohan Ajit, Koustav Acharya und Abhishek Samanta. „A review of convolutional neural networks“. In: *2020 international conference on emerging trends in information technology and engineering (ic-ETITE)*. IEEE. 2020, S. 1–5.

[3] Saad Albawi, Tareq Abed Mohammed und Saad Al-Zawi. „Understanding of a convolutional neural network“. In: *2017 International Conference on Engineering and Technology (ICET)*. 2017, S. 1–6. DOI: 10.1109/ICEngTechnol.2017.8308186.

[4] Ghassan Mahmoud Husien Amer und Ahmed Mohamed Abushaala. „Edge detection methods“. In: *2015 2nd World Symposium on Web Applications and Networking (WSWAN)*. IEEE. 2015, S. 1–7.

[5] Holger Aust. „Das Gehirn kopieren? – Künstliche neuronale Netze“. In: *Das Zeitalter der Daten: Was Sie über Grundlagen, Algorithmen und Anwendungen wissen sollten*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2021, S. 161–193. ISBN: 978-3-662-62336-7. DOI: 10.1007/978-3-662-62336-7_6. URL: https://doi.org/10.1007/978-3-662-62336-7_6.

[6] Arian Azarang, Hafez E Manoochehri und Nasser Kehtarnavaz. „Convolutional autoencoder-based multispectral image fusion“. In: *IEEE access* 7 (2019), S. 35673–35683.

[7] Klaus Backhaus, Bernd Erichson und Rolf Weiber. „Neuronale Netze“. In: *Fortgeschrittene Multivariate Analysemethoden: Eine anwendungsorientierte Einführung*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, S. 295–347. DOI: 10.1007/978-3-662-46087-0_6. URL: https://doi.org/10.1007/978-3-662-46087-0_6.

[8] Chrislb. *Künstliches Neuron*. 2023. URL: https://de.wikipedia.org/wiki/K%C3%BCnstliches_Neuron.

[9] Bin Ding, Huimin Qian und Jun Zhou. „Activation functions and their characteristics in deep neural networks“. In: *2018 Chinese Control And Decision Conference (CCDC)*. 2018, S. 1836–1841. DOI: 10.1109/CCDC.2018.8407425.

[10] Alexandru Dinu. 2021. URL: https://github.com/alexandru-dinu/cae/blob/master/src/models/cae_16x8x8_zero_pad_bin.py (besucht am 2023).

[11] Wissal Farsal, Samir Anter und Mohammed Ramdani. „Deep learning: An overview“. In: *Proceedings of the 12th International Conference on Intelligent Systems: Theories and Applications*. 2018, S. 1–6.

[12] Wenshuo Gao u. a. „An improved Sobel edge detection“. In: *2010 3rd International conference on computer science and information technology*. Bd. 5. IEEE. 2010, S. 67–71.

[13] Ross Girshick u. a. „Rich feature hierarchies for accurate object detection and semantic segmentation“. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014, S. 580–587.

[14] Xavier Glorot und Yoshua Bengio. „Understanding the difficulty of training deep feedforward neural networks“. In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop und Conference Proceedings. 2010, S. 249–256.

[15] Xavier Glorot, Antoine Bordes und Yoshua Bengio. „Deep sparse rectifier neural networks“. In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. JMLR Workshop und Conference Proceedings. 2011, S. 315–323.

[16] Caglar Gulcehre u. a. „Noisy Activation Functions“. In: *Proceedings of The 33rd International Conference on Machine Learning*. Hrsg. von Maria Florina Balcan und Kilian Q. Weinberger. Bd. 48. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, 2016, S. 3059–3068. URL: https://proceedings.mlr.press/v48/gulcehre16.html.

[17] Kaiming He u. a. „Deep residual learning for image recognition“. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, S. 770–778.

[18] Kaiming He u. a. „Delving deep into rectifiers: Surpassing human-level performance on imagenet classification“. In: *Proceedings of the IEEE international conference on computer vision*. 2015, S. 1026–1034.

[19] Christian Janiesch, Patrick Zschech und Kai Heinrich. „Machine learning and deep learning“. In: *Electronic Markets* 31.3 (2021), S. 685–695.

[20] David Kriesel. *Ein kleiner Überblick über Neuronale Netze*. 2007. URL: erh%5C%22%7Ba%7Dltlich%20auf%20http://www.dkriesel.com.

[21] Alex Krizhevsky, Ilya Sutskever und Geoffrey E Hinton. „Imagenet classification with deep convolutional neural networks“. In: *Communications of the ACM* 60.6 (2017), S. 84–90.

[22] Yann Lecun. *PhD thesis: Modeles connexionnistes de l'apprentissage (connectionist learning models)*. English (US). Universite P. et M. Curie (Paris 6), Juni 1987.

[23] Yann LeCun u. a. „Gradient-based learning applied to document recognition“. In: *Proceedings of the IEEE* 86.11 (1998), S. 2278–2324.

[24] Lu Lu u. a. „Dying relu and initialization: Theory and numerical examples“. In: *arXiv preprint arXiv:1903.06733* (2019).

[25] Qinxue Meng u. a. „Relational autoencoder for feature extraction“. In: *2017 International joint conference on neural networks (IJCNN)*. IEEE. 2017, S. 364–371.

[26] Keiron O'Shea und Ryan Nash. „An introduction to convolutional neural networks“. In: *arXiv preprint arXiv:1511.08458* (2015).

[27] Prajit Ramachandran, Barret Zoph und Quoc V Le. „Searching for activation functions“. In: *arXiv preprint arXiv:1710.05941* (2017).

[28] Andrinandrasana David Rasamoelina, Fouzia Adjailia und Peter Sinčák. „A Review of Activation Function for Artificial Neural Network“. In: *2020 IEEE 18th World Symposium on Applied Machine Intelligence and Informatics (SAMI)*. 2020, S. 281–286. DOI: 10.1109/SAMI48414.2020.9108717.

[29] Joseph Redmon u. a. „You only look once: Unified, real-time object detection“. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, S. 779–788.

[30] Anh H. Reynolds. *Convolutional Neural Networks (CNNs)*. 2019. URL: https://anhreynolds.com/blogs/cnn.html (besucht am 2023).

[31] Jurgen Schmidhuber, U Meier und D Ciresan. „Multi-column deep neural networks for image classification“. In: *2012 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE Computer Society. 2012, S. 3642–3649.

[32] Sefik Ilkin Serengil. *Convolutional Autoencoder: Clustering Images with Neural Networks*. 2018. URL: https://sefiks.com/2018/03/23/convolutional-autoencoder-clustering-images-with-neural-networks/ (besucht am 2023).

[33] Pierre Sermanet u. a. „Overfeat: Integrated recognition, localization and detection using convolutional networks“. In: *arXiv preprint arXiv:1312.6229* (2013).

[34] Sagar Sharma, Simone Sharma und Anidhya Athaiya. „Activation functions in neural networks“. In: *Towards Data Sci* 6.12 (2017), S. 310–316.

[35] M Traeger u. a. „Künstliche neuronale Netze: Theorie und Anwendungen in der Anästhesie, Intensiv-und Notfallmedizin“. In: *Der Anaesthesist* 52 (2003), S. 1055–1061.

[36] Michael Tschannen, Olivier Bachem und Mario Lucic. „Recent advances in autoencoder-based representation learning“. In: *arXiv preprint arXiv:1812.05069* (2018).

[37] O Rebecca Vincent, Olusegun Folorunso u. a. „A descriptive algorithm for sobel image edge detection“. In: *Proceedings of informing science & IT education conference (InSITE)*. Bd. 40. 2009, S. 97–107.

[38] Pascal Vincent u. a. „Extracting and composing robust features with denoising autoencoders“. In: *Proceedings of the 25th international conference on Machine learning*. 2008, S. 1096–1103.

[39] Antónia Vojteková. „Neural network noise reduction of astronomical images“. In: (2020), S. 4.

[40] Zhou Wang u. a. „Image quality assessment: from error visibility to structural similarity“. In: *IEEE transactions on image processing* 13.4 (2004), S. 600–612.

[41] Chathurika S. Wickramasinghe, Daniel L. Marino und Milos Manic. „ResNet Autoencoders for Unsupervised Feature Learning From High-Dimensional Data: Deep Models Resistant to Performance Degradation“. In: *IEEE Access* 9 (2021), S. 40511–40520. DOI: 10.1109/ACCESS.2021.3064819.

[42] Alireza Zaeemzadeh, Nazanin Rahnavard und Mubarak Shah. „Norm-preservation: Why residual networks can become extremely deep?“ In: *IEEE transactions on pattern analysis and machine intelligence* 43.11 (2020), S. 3980–3990.

[43] Matthew D Zeiler und Rob Fergus. „Visualizing and understanding convolutional networks“. In: *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I 13*. Springer. 2014, S. 818–833.

[44] Yifei Zhang. „A better autoencoder for image: Convolutional autoencoder“. In: *ICONIP17-DCEC. Available online: http://users. cecs. anu. edu. au/Tom. Gedeon/conf/ABCs2018/paper/ABCs2018_paper_58. pdf (accessed on 23 March 2017)*. 2018.