Convolutional Auto Encoder For Image Denoising

Tobias Weinert
Fakultät Elektrotechnik, Medizintechnik und Informatik
Angewandte Informatik
Hochschule Offenburg
Badstraße 24, 77652 Offenburg
tweinert@stud.hs-offenburg.de

***Abstract*— In dieser wissenschaftlichen Arbeit werden die Grundlagen von neuronalen Netzen und Autoencodern mit den wichtigsten mathematischen Hintergründen dargelegt. Ziel dieser Arbeit ist es außerdem, ein konkretes Anwendungsbeispiel zu implementieren. Insbesondere der Aufbau künstlicher Neuronen und der Zusammenschluss dieser zu einem gefalteten neuronalen Netz, in diesem Fall einem Convolutional Autoencoder (CAE), wird erläutert. Prozesse wie die Konvolution, Pooling und Upsampling werden näher untersucht. Die Wahl des Convolutional Autoencoders für die Problemstellung der Rauschentfernung wird begründet und die Verarbeitung von Bilddateien durch dieses Netz wird näher beschrieben. Mit diesem Wissen wird das Aufsetzen und Trainieren einer künstlichen Intelligenz in Python erläutert, welche mittels Daten darauf hin trainiert wird, das Rauschen (*noise*) von Bildern zu entfernen. Hierzu wird die Python-Library Keras und der MNIST Trainingsdatensatz verwendet. Die von diesem Netz erzielten Ergebnisse werden im Abschluss analysiert.**

***Stichworte*— Convolutional Autoencoder, Künstliche Intelligenz, Denoising, Neuronale Netze, Bildverarbeitung, Keras**

I. EINFÜHRUNG

Der Prozess der Rauschentfernung bzw. des Entrauschens von Bilddateien (*image denoising*) ist ein fundamentales Gebiet der Computer Vision. Dieses Gebiet beschäftigt sich mit der Verarbeitung und Analyse von Bildern und findet vielfältige Anwendungsfälle, wie die Wiederherstellung, Komprimierung und Verbesserung von Bilddateien. In der Realität werden diese Techniken beispielsweise bei Scans von medizinischen Geräten verwendet, um klarere Bilder zu erhalten und bessere Diagnosen stellen zu können. Gerade hier wird die Notwendigkeit für zuverlässliche Entrauschungsverfahren klar.

Es gibt verschiedene Ansätze für die Rauschentfernung von Bildern. Zum einen gibt es klassische Ansätze, welche ohne künstliche Intelligenz auskommen. Ein Prominentes Beispiel ist hier die Wavelet-basierte Entrauschungsmethode, welche ein Frequenzband anhand eines Signals, was in diesem Falle das Bild ist, anlegt und so den hochfrequentierten Noise herausfiltern kann. Während diese Techniken viele gute Charakteristiken besitzen und auf spezifische Daten angepasst werden können, versagen diese schnell bei komplexen und unterschiedlichen Daten. Sie sind nicht adaptiv und passen sich nicht dem Kontext des Bildes an, da die zusätzlich benötigten Parameter für jeden Datensatz sorgfältig bestimmt und abgewogen werden müssen. [1]

Für komplexere Anwendungsfälle wird also ein adaptiver Ansatz benötigt. Das System muss sich den Daten anpassen und auf unvorhergesehenes Rauschen autonom reagieren können. In den letzten Jahren hat das Gebiet der künstlichen Intelligenz enorme Fortschritte erreicht und hielt auch in der Computer Vision Einzug. Convolutional Autoencoder sind weit verbreitete Deep-Learning-Modelle, welche häufig für das Problem der Rauschentfernung verwendet werden. Das Ziel ist es, diesen Autoencoder mit Daten so zu trainieren, dass dieser das Rauschen von Bildern erkennen und so entfernen kann, dass ein möglichst verlust- und rauschfreies Ergebnisbild entsteht.

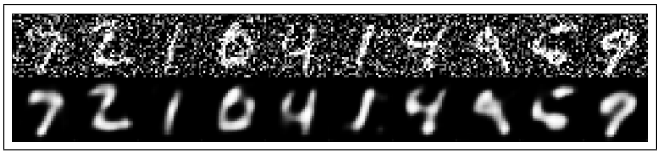

Fig. 1. In- und Output des Autoencoders (Matplotlib)

Das in Fig. 1 zu sehende Bild zeigt den erwünschten entrauschten Output bei einem verrauschten Input. Im Input ist ein deutliches additives weißes gaußsches Rauschen zu sehen, welche von dem Autoencoder entfernt werden soll. Während das Rauschen weitestgehend reduziert werden soll, ist es dennoch wichtig, dass Informationen wie Kanten und Muster des Bildes weitestgehend erhalten bleiben.

Dieses Paper ist wie folgt strukturiert: Sektion II führt in die Thematik der künstlichen Neuronen und Netzwerke ein. Sektion III erläutert Convolutional Neural Networks

und die benötigten mathematischen Grundlagen hierfür. In Sektion IV wird dann die Architektur des Autoencoders erklärt, welcher in Sektion V mit Hilfe von Python und Keras entwickelt wird. Dieser Autoencoder wird auf das Denoising von Bilddateien trainiert.

II. NEURONALE NETZE

A. Problemstellung

Künstliche Intelligenz beruht auf vielen Bereichen der Mathematik. Insbesondere sind lineare Algebra, Analysis, Stochastik und Statistik Kerngebiete dieser Technologie. Um das Modell des Convolutional Autoencoder verstehen zu können, müssen einige fundamentale mathematische Konzepte erläutert werden. Diese bilden die Basis der Architektur eines CAE und sind essentiell, um das Modell bauen und trainieren zu können.

Mathematisch gesehen kann das Problem des Rauschens relativ leicht ausgedrückt werden. Es sei das Signal bzw. Bild gegeben durch $f(x,y)$ mit den X- und Y-Koordinaten (x,y). Das zu entfernende Rauschen sei definiert durch $n(x,y)$. Dann lässt sich das verrauschte Bild wie folgt definieren:

$$f'(x,y) = f(x,y) + n(x,y) \tag{1}$$

Ziel ist es nun, eine künstliche Intelligenz zu entwickeln, welche das rauschfreie Bild

$$f(x,y) = f'(x,y) - n(x,y) \tag{2}$$

ermitteln kann. Wie sich herausstellte, sind gerade Autoencoder besonders gut darin, diesen Noise mit möglichst geringem Verlust zu entfernen [2].

B. Künstliche Neuronen

Unter künstlichen neuronalen Netzwerken versteht man das technische Nachbilden der Funktionsweise des menschlichen Gehirns. Sie sind Systeme, die ihren internen Zustand an sich ändernde Bedingungen anpassen können. Sie kommen bei nicht-linearen Problemen zum Einsatz, also dann, wenn es keinen festen Lösungsweg gibt und dieser erst gefunden werden muss [3].

Ähnlich wie das menschliche Gehirn bestehen künstliche Netze aus Knoten, welche auch processing Elements (PE) genannt. Jeder dieser Knoten besitzt einen Input und einen Output. Dies kann jeweils ein anderer Knoten, aber auch die Umgebung selbst sein, aus der die Trainingsdaten kommen. Ebenfalls besitzt jedes processing Element auch eine Funktion f, welche die durch den Input erhaltenen Daten auf eine gewisse Weise transformiert und letztendlich über den Output weiter gibt. Die Verbindungen zwischen den Knoten werden über Zahlenwerte klassifiziert. Üblicherweise bedeutet ein positiver Wert eine Anregung bzw. ein erhöhtes Aktivitätslevel des Elements. Dies hat zur Folge, dass die Funktion des Knotens einen höheren Einfluss auf das Gesamtergebnis des Netzes hat. Demzufolge wird diese Klassifizierung auch "Gewicht" genannt. Ein negativer Wert des Gewichts bedeutet, dass eine hemmende Verbindung vorliegt und der Knoten weniger bzw. keinen Einfluss auf das Netz hat [3].

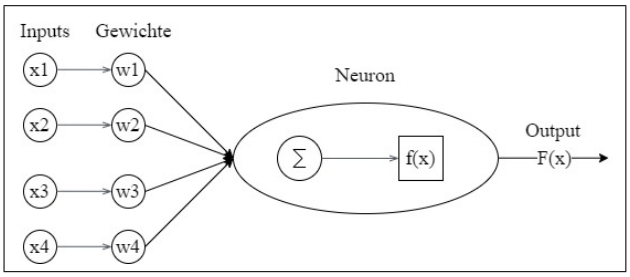

Fig. 2. Aufbau künstliches Neuron

In Fig. 2 sind zusätzlich die Summen- und Aktivierungsfunktion zu sehen, welche im Kern des processing Elements liegen und zusammen das künstliche Neuron bilden.

Die Summenfunktion ist eine mathematische Operation, welche dann durchgeführt wird, wenn das Neuron i durch den Input ein Signal von seinen eingehenden Verbindungen erhält. Jedes einkommende Signal wird mit der Gewichtung w der Verbindung multipliziert und dann an die Summenfunktion weitergegeben, um das Aufsummieren der gewichteten Verbindungen durchzuführen.

$$\sum_{j=1}^{n} w_{ij} x_j \tag{3}$$

Die Aktivierungsfunktion wendet dann eine Funktion $f(x)$ auf das Ergebnis der Summenfunktion an. Das Ergebnis x_i ist somit

$$x_i = f\left(\sum_{j=1}^{n} w_{ij} x_j\right) \tag{4}$$

und ist die Ausgabe des Neurons. Es gibt eine Vielzahl an Aktivierungsfunktionen, im Kontext des Autoencoders werden jedoch lediglich die Sigmoidfunktion und ReLU benötigt. Die Sigmoidfunktion ist eine weit verbreitete Aktivierungsfunktion und ist wie folgt definiert:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{5}$$

Wie aus Fig. 3 hervorgeht, besitzt die Sigmoidfunktion einen Wertebereicht von $[0;1]$ und ist somit für stetige Neuronen geeignet. Aufgrund ihrer einfachen Differenzierbarkeit wird diese häufig als Aktivierungsfunktion verwendet und ermöglicht so Lernmechanismen wie den Backpropagation-Algorithmus, welcher später weiter erläutert wird [4]. Eine weitere Aktivierungsfunktion ist die

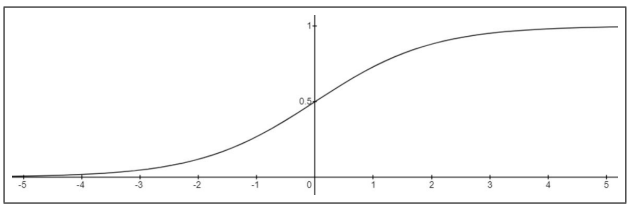

Fig. 3. Graph der Sigmoidfunktion

in Fig. 4 zu sehende ReLU und bedeutet *rectified linear Unit*. Diese ist definiert als

$$f(x) = \max\{0, x - \theta\} \tag{6}$$

und nimmt einen skalaren Input x. Der Schwellwert θ legt fest, ab welchem Punkt die Funktion aktiv wird. Wenn der Input abzüglich θ größer oder gleich Null ist, gibt die Funktion den Input Wert zurück.

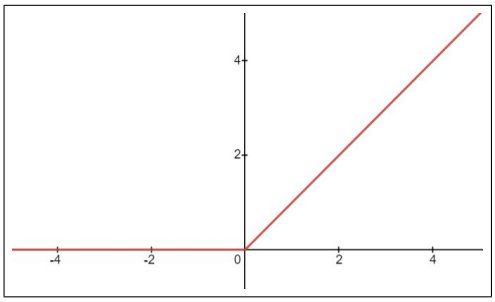

Fig. 4. Graph der ReLU

Ist der Eingabewert abzüglich θ jedoch kleiner als Null, ergibt sie Null. Daraus folgt, dass die ReLU eine einfache lineare Funktion für positive Eingaben, und konstant Null für negative Eingaben ist. Dies hat zur Folge, dass sie rechnerisch effizient ist und sich besonders gut als Aktivierungsfunktion eignet [5].

C. Künstliche neuronale Netze

Die Verbindungen zwischen den Knoten können sich im Laufe der Zeit eigenständig modifizieren. Durch das mehrmalige Eingeben der Daten entsteht ein Lernprozess, welcher das künstliche neuronale Netz immer weiter an die Gegebenheiten der Umgebung anpasst.

Um ein solches Netz zu bilden, werden die künstlichen Neuronen in ein- oder mehrdimensionale Schichten gruppiert. Der Grad der Dimension ist hierbei abhängig von der Qualität und Menge der in das Netz eingegebenen Daten. Üblicherweise wird eine feed-forward Topologie für den Aufbau eines solchen Netzes verwendet. Hierzu wird eine gewisse Anzahl an processing Elements gruppiert und als Input-Schicht verwendet. Wie der Begriff "feed-forward" andeutet, können Daten über die Verbindungen immer nur nach vorne weiter gereicht werden und es dürfen in der Topologie des Netzwerks keine Zyklen auftreten. Die Input-Schicht gibt in diesem Fall die Informationen an eine oder mehrere Zwischenschichten, die sogenannten "hidden Layers", weiter. Dort werden die Daten weiter durch in dieser Schicht befindlichen PEs verarbeitet und an die Output-Schicht weitergegeben. Die Anzahl der Neuronen in dieser Schicht ist abhängig von dem zu lösenden Problem. Bei einem Problem mit einer binären Lösung würde ein einziges Element ausreichen. Für das Problem der Rauschentfernung aus Bildern werden mehrere Neuronen in dieser Schicht benötigt. Jedes dieser Neuronen produziert dann einen Teil des Ausgabebildes.[3].

In Fig. 4 ist ein solches dreilagiges, vorwärtsgerichtetes Netz dargestellt.

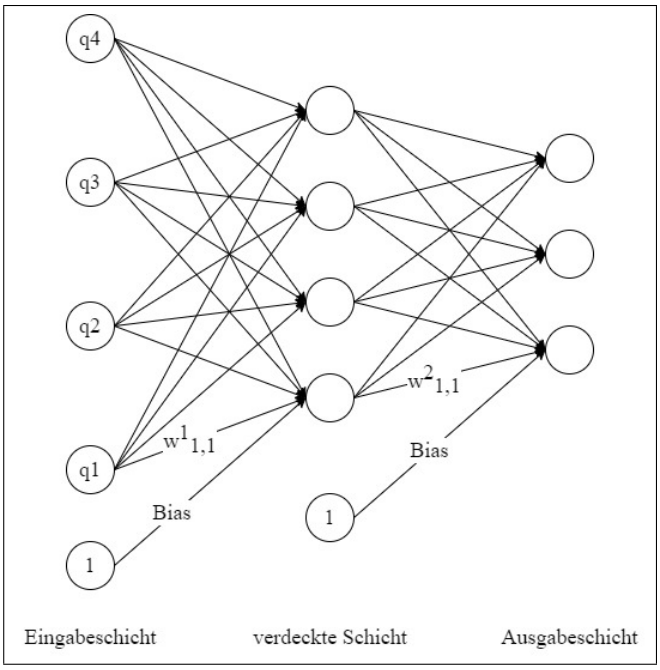

Fig. 5. Topologie eines dreilagigen feed-forward Netzes

Zu erkennen ist in Fig. 4 ebenfalls, dass das das hinterste Neuron der Eingabe- bzw. Verdeckten-Schicht ein Bias-Neuron ist. Dieses wird einem konstanten Wert von Eins initialisiert und die Gewichtung dieser Neuronen werden ebenfalls gelernt.

Dieses Netz muss nun trainiert werden. Grundsätzlich gibt es vier verschiedene Arten, die diesen Prozess durchzuführen.

- Supervised Learning: Das Netz lernt durch vorgegebene Trainingsdaten. Diese bestehen aus dem Input und dem gewünschten Output. Das Netz berechnet dann die Fehlerabweichung zum gewünschten Ergebnis und kann so die Verlustfunktion optimieren.
- Unsupervised Learning: Hier wird dem Netz keine Lösung vorgegeben, nach der es sich optimieren kann. Stattdessen erkundet der Algorithmus selbstständig die Daten und versucht aus diesen Muster und Zusammenhänge abzuleiten.
- Reinforcement Learning: Nach jeder Iteration des Algorithmus wird dem System lediglich mitgeteilt, ob es das gewünschte Ergebnis erreicht hat. Dem Netz wird hierbei nicht mitgeteilt, in welche Richtung es sich entwickeln muss.
- Backpropagation Learning: Hierbei handelt es sich um eine spezielle Form des supervised Learning, welche besonders geeignet für das Entrauschen von Bilddateien bzw. Autoencodern geeignet ist [6].

Fügt man an das in Fig. 3 gezeigte Drei-Schichten-Netz noch eine vierte Schicht, die Zielausgabe an, erhält man ein Backpropagation-Netz. In der Zielausgabeschicht werden die zum derzeitigen Trainingsstandes vom Netz produzierten Werte der Ausgabeschicht mit den Werten

vergleichen, die dieses erreichen soll. Auf diese Weise lernt das Netz einen Zustand zu erreichen, welcher möglichst nahe zu dem von der Zielausgabeschicht vorgegebenen liegt. Hierzu durchläuft das Netz mittels dem Backpropagation-Algorithmus mehrere Iterationen. [5].

Algorithmus 1: Backpropagation-Algorithmus

Input : Netzwerkeingaben x, Zielausgaben y
Output: Aktualisierte Gewichtungen
Initialisiere Gewichtungen zufällig
while *nicht konvergiert* **do**
  for *jedes Trainingsbeispiel* (x, y) **do**
    - Propagiere Input nach vorne durch das Netz
    - Berechne Aktivationslevel jedes Layers mit den aktuellen Gewichtungen
    - Berechne das quadratischen Fehler mit der Zielausgabeschicht
    - Propagiere den Fehler zurück an die vorherigen Schichten
    for *jeder Layer l in umgekehrter Reihenfolge* **do**
      - Multipliziere Fehler mit Ableitung der Aktivationsfunktion
    end
    for *jedes Gewicht w* **do**
      - Aktualisiere die Gewichtung jedes Layers
    end
    Berechne den durchschnittlichen Fehler für die Trainingsbeispiele
  end
end

III. Convolutional Neural Networks

Das Konzept eines künstlichen neuronalen Netzwerks, auch MLP (multilayer perceptron) genannt, wurde bereits vorgestellt. Ein *Convolutional Neural Network* erweitert diese MLPs in dem Sinne, dass in den verdeckten Schichten weitere Layer eingefügt werden. Eine dieser Schichten ist der Convolutional Layer. Es hat sich herausgestellt, dass CNNs besonders gut für die Analyse von Bilddateien sind, jedoch werden diese auch für andere Problemstellungen eingesetzt. Besonders ihre Stärken in der Mustererkennung machen diese so beliebt in dem Gebiet der Computer Vision [7].

A. Architektur

In Fig. 6 ist eine mögliche Architektur eines CNNs zu sehen. Diese können sich anhand der Schichten unterscheiden, durch die das Netzwerk gebildet wird. Alle CNNs haben jedoch gemeinsam, dass sie mindestens eine Konvolutionsschicht besitzen. In Fig. 5 sind noch weitere Schichten wie der Pooling Layer zu sehen, welche später weiter erläutert werden.

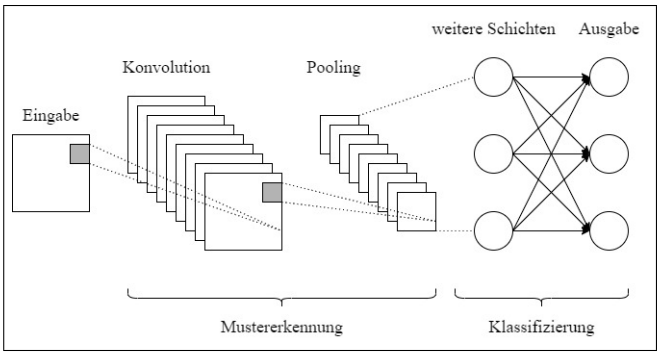

Fig. 6. Architektur eines Convolutional Neural Network

Ebenfalls zu sehen ist, dass es sich wie bei MLPs um ein feed-forward Netz handelt, welches einen Input besitzt, der die Daten in das Netz leitet und am Ende einen Output erzeugt.

B. Mathematische Grundlagen

Um den Prozess der Mustererkennung zu demonstrieren, wird im späteren Verlauf ein einfaches praktisches Beispiel eingeführt. Hierfür sind einige mathematische Grundlagen notwendig.

Ein Vektor wird mit einem Kleinbuchstaben, beispielsweise $x \in \mathbb{R}^D$, angegeben. Hierbei ist $\mathbb{R}$ die Menge der reellen Zahlen. Dieser Vektor kann auch als eine Spalte mit D Elementen bzw. Zeilen gesehen werden. Mit einem Großbuchstaben, wie z. B. $X \in \mathbb{R}^{H \times W}$ werden Matrizen angegeben, welche H Reihen und W Spalten besitzen. Matrizen dritter Ordnung, wie etwa $X \in \mathbb{R}^{H \times W \times D}$, werden auch Tensor dritter Ordnung genannt. Ein solcher Tensor kann als D Schichten oder Channeln von $H \times W$ Matrizen gesehen werden, was in Fig. 6 als Convolutional Layer visualisiert ist. Geht man von einem Bild im RGB-Format aus, so kann dieses als Tensor dritter Ordnung repräsentiert werden. Sei die Höhe H und die Breite W des Bildes in Pixeln, so entspricht das Bild einem Tensor $B \in \mathbb{R}^{H \times W \times 3}$. Visuell gleicht das einem Stapel aus drei $H \times W$ Matrizen. Jede dieser Matrizen hält die Information über die Farbintensität von Rot, Grün oder Blau für einen Pixel. Ein einzelner Pixel kann hierbei mit dem Tripel (i, j, d) mit $0 \leq i < H$, $0 \leq j < W$, und $0 \leq d < D$ indexiert werden. Während gewöhnliche ANNs nur Matrizen verarbeiten können und jedes Bild in Graustufen konvertieren müssen, können CNNs über diese Tensoren die Farben beibehalten und diese im Lernprozess berücksichtigen.

Das Skalar- bzw. Kreuzprodukt aus zwei Matrizen ist ein binärer Operator, welcher zwei Matrizen gleicher Dimensionen in einen skalaren Wert umwandelt. Seien zwei Matrizen $A \in \mathbb{R}^{n \times m}$ und $B \in \mathbb{R}^{n \times p}$, so ist das Kreuzprodukt dieser definiert als:

$$A \times B = \sum_{k=1}^{n} A_{ik} B_{kj} = A_{0,0} \cdot B_{0,0} + A_{0,1} \cdot B_{0,1} + \dots \qquad (7)$$

Die Konvolution der Matrizen A und B mit $m \geq p$ und

$n \geq q$ kann mit der Verwendung des Kreuzprodukts als

$$(A * B)i, j = \sum k = 1^n \sum_{l=1}^{p} A_{i-k+1, j-l+1} B_{k,l} \tag{8}$$

definiert werden. Es ist anzumerken, dass (8) zur Vereinfachung eine Schrittweite von eins verwendet. Dieser Operator wird auch Faltungsoperator genannt. Es ist hierbei notwendig, dass die zweite Matrix dimensional kleiner ist, als die erste.

Die Konvolution zweier Matrizen entspricht visuell dem Übereinanderlegen und der Anwendung des Kreuzprodukts auf die überlappenden Matrizen. Dann wird die Matrix B eine Schrittweite nach rechts geschoben und erneut das Kreuzprodukt gebildet. Die berechneten Skalare werden aufaddiert und in die Ergebnismatrix eingetragen, bis die zweite Matrix am rechten Rand angelangt ist. Dann wird diese zweite Matrix, auch Filtermatrix genannt, wieder ganz nach links und eine Reihe nach unten verschoben, bis die gesamte Eingabematrix abgearbeitet ist [7].

Mit diesen Grundlagen kann nun anhand eines Beispiels der Prozess der Konvolution eines Bildes mit einem Filter demonstriert werden.

C. Convolutional Layers

Die Convolutional Layer sind für die Mustererkennung zuständig. Oft sind mehrere dieser Layer hintereinandergeschalten und werden mit jedem Layer spezifischer. So erkennen die ersten Schichten meist nur einfache Kanten, spätere Schichten dann geometrische Formen und komplexe Muster wie beispielsweise Gesichter. Eine gewisse Tiefe dieser Layer ist notwendig, um zuverlässige Ergebnisse für komplexe Datensätze zu erzielen. Jede weitere Schicht benötigt eine weitere Konvolutionsoperation, was die Inferenzzeit, also die totale Zeit des Durchlaufens der Daten durch das Netz, erhöht. Jedoch kann schon das Entfernen eines einzelnen Layers starke Einbußen in der Genauigkeit des Netzwerks bedeuten [8]. In jedem der Convolutional Layer befindet sich ein Filter, auch Kernel genannt. Hierbei handelt es sich um einen Tensor, welcher dieselbe Tiefe wie das Eingabebild besitzt.

Um die Konvolution besser darstellen zu können, wird im Folgenden ein Beispiel eingeführt, welches diesen Prozess visualisiert. Ein Bild wird in die Eingabeschicht, wie in Fig. 6 zu sehen war, in das Netz eingegeben. In diesem Beispiel wird das Bild der Zahl Acht in Graustufen dargestellt. Hierzu wird jedem Pixel ein Wert von 0-1 zugeordnet. Ein Wert von 0 entspricht dabei Dunkelgrau, ein Wert von 1 der Farbe Weiß. In Fig. 7 ist die zugehörige Matrix zu sehen.

Ziel ist es nun, eine einfache Kantenerkennung an dem Bild durchzuführen. Um die oberen Kanten der Ziffer Acht zu erkennen, wird ein sogenannter "Top-Edge-Filter" verwendet, welcher durch die Matrix

$$F = \begin{bmatrix} -1.0 & -1.0 & -1.0 \\ 1.0 & 1.0 & 1.0 \\ 0.0 & 0.0 & 0.0 \end{bmatrix} \tag{9}$$

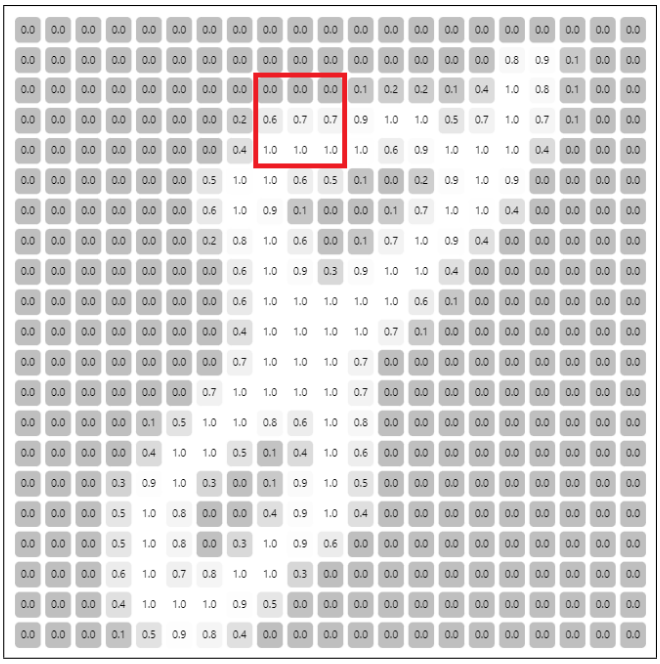

Fig. 7. Zahl 8 als Graustufen-Matrix visualisiert

repräsentiert wird. Es wird also in jedem Iterationsschritt der Konvolution ein 3×3 Ausschnitt des Eingabebildes betrachtet und das Skalarprodukt mit dem Kernel gebildet. Aus Veranschaulichungszwecken wird davon ausgegangen, dass sich die Konvolution bereits in einem späteren Schritt befindet. Für den in Fig. 7 markierten Bereich lautet der Eintrag in der Ergebnismatrix des Iterationsschritte nach (8) dann:

$$\begin{aligned} E_{3,6} = & \begin{bmatrix} -1.0 & -1.0 & -1.0 \\ 1.0 & 1.0 & 1.0 \\ 0.0 & 0.0 & 0.0 \end{bmatrix} \times \begin{bmatrix} 0.0 & 0.0 & 0.0 \\ 0.6 & 0.7 & 0.7 \\ 1.0 & 1.0 & 1.0 \end{bmatrix} \\ & = 0.0 \cdot -1.0 + 0.0 \cdot -1.0 \\ & \quad + 0.0 \cdot -1.0 + 0.0 \cdot -1.0 + 0.0 \cdot -1.0 \\ & \quad + 0.6 \cdot 1.0 + 0.7 \cdot 1.0 + 0.7 \cdot 1.0 \\ & \quad + 1.0 \cdot 0.0 + 1.0 \cdot 0.0 + 1.0 \cdot 0.0 + 1.0 \cdot 0.0 = 2.0 \end{aligned} \tag{10}$$

Nach der Berechnung wird das Fenster des Kernels auf der Eingabematrix um die Schrittweite 1 nach rechts verschoben und das nächste Kreuzprodukt aus dem Ausschnitt der Eingabematrix und dem Kernel berechnet. Führt man diesen Algorithmus bis zum Ende der Eingabematrix fort, so entsteht Fig. 8.

Die Werte der Matrixeinträge entsprechen einem Farbgradienten von Dunkelblau (negativ) bis Dunkelrot (positiv). In dieser Matrix, auch Feature Map genannt. ist deutlich zu erkennen, wie durch den Top-Edge-Filter die oberen Kanten der Zahl Acht erkannt wurde. Die negativen, Blau markierten Werte entsprechen genau dem inversen des verwendeten Filters, in diesem Fall den unteren Kanten. Da das Ziel war, lediglich die oberen Kanten zu erkennen, müssen diese Werte herausgefiltert werden. Dies ist möglich, indem die bereits vorgestellte ReLU als

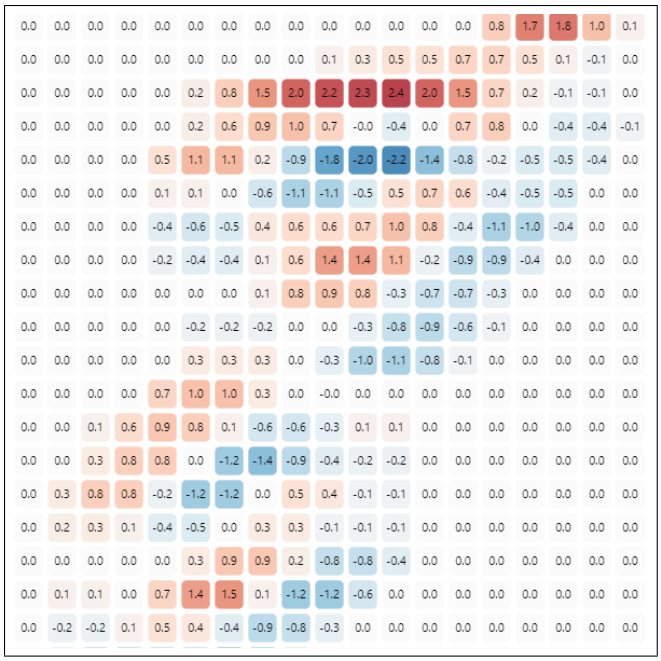

Fig. 8. Ergebnismatrix

Aktivierungsfunktion verwendet wird. Unter Verwendung dieser Funktion werden die positiven Werte beibehalten und die negativen auf null gesetzt. Würde zusätzlich eine Zielausgabeschicht implementiert werden, so würde dann der Backpropagation-Algorithmus ablaufen und sich weiter dem gewünschten Ergebnis annähern. Bemerklich ist, dass durch relativ einfache Mathematik eine Form in einem Bild erkannt werden konnte.

Durch die Verwendung von Graustufen und folglich das Entfallen mehrerer Channel wurde das Beispiel dimensional stark vereinfacht. Die Konvolution mit farbigen Eingabebildern lässt sich jedoch mit (7) und (8) auf dieselbe Art durchführen. Hierzu würde lediglich die einfache Graustufen-Matrix durch einen Tensor dritter Ordnung ausgetauscht werden, dann kann der Konvolutionsoperator auf den Tensor mit den Filtern angewendet werden. Der Filter läuft dann also über jeden Channel einzeln. Jeder weitere dieser Input Channel sorgt jedoch für steigende Komplexität des Konvolutionsoperators und erhöht damit die Laufzeit [9].

Wie bereits erwähnt würden diesem hier vorgestellten Filter noch weitere folgen, um komplexere Muster erkennen zu können, jedoch wurde das Konzept der Mustererkennung deutlich dargestellt.

D. Upsampling

Upsampling ist quasi Konvolution in umgekehrter Abfolge und wird *transponierte Konvolution* genannt. Diese mathematische Operation wird für den Autoencoder benötigt und deshalb hier kurz erklärt. Bei Upsampling wird ein Set aus gelernten Kerneln bzw. Filtern auf den Input angewendet, jedoch statt diesen wie bei der Konvolution dimensional zu verkleinern, wird dieser vergrößert. Jeder skalare Eintrag aus der Eingabematrix wird mit dem Kernel multipliziert und so entsteht aus einem Skalar mit einem Kernel $K = \mathbb{R}^{H \times W}$ $H \times W$ Einträge in der Ausgabematrix [7].

E. Pooling Layers

Wie bereits erwähnt führt das Hinzufügen von Input Channeln zu einem höheren Rechenaufwand für das Netzwerk. Um diesem Umstand entgegen zu wirken, werden Pooling Layers verwendet. Diese befinden sich immer hinter den Convolutional Layers. Die Eingabedaten in die Poolingschicht sind somit die Ausgabedaten der Konvolutionsschicht. Dieser ist dafür verantwortlich, die räumlichen Dimensionen, also die Menge der Eingabedaten, zu verkleinern. Folglich kann durch diese Schicht Rechenzeit und Speicherplatz eingespart werden. Gewöhnlicherweise erhält dieser Layer einen Tensor dritter Ordnung als Eingabe und produziert ebenfalls einen Tensor dritter Ordnung als Ausgabe, bestehend aus Höhe, Breite und Tiefe. Hierbei ist die Höhe und Breite nach dem Durchlaufen der Poolingschicht kleiner als zuvor.

Der Pooling Layer arbeitet auf jedem Channel einzeln und reduziert dort die Höhe und Breite jedes Channels. Hierfür wird die Pooling Operation angewendet, welche sehr ähnlich zu der bereits erläuterten Konvolutionsoperation ist. Es existieren verschiedene Arten von Pooling, Max Pooling ist jedoch die meist verbreitete. Es wird ähnlich wie bei der Konvolution ein Fenster über das Eingabebild gelegt. Dieses umfasst eine gewisse Höhe und Breite, auch Pool Size genannt. Alle Werte des Bildes in dem Pool werden verglichen, und der höchste Wert wird in die Ausgabematrix übernommen. Die Ausgabematrix wird also dimensional umso kleiner, je höher die Pool Size ist. Eine weitere, weniger häufige Variante ist das Average Pooling. Es wird anstatt des Maximums also ein Durchschnitt der in dem Pool befindlichen Werte genommen und in die Ergebnismatrix eingetragen.

Es kann zwar sinnvoll sein, Pooling Layer zu verwenden, jedoch sollte dabei abgewogen werden, wie stark das Pooling ist und ob der Verlust von Details für das gewünschte Ergebnis in diesem Maß akzeptabel ist [10].

F. Fully Connected Layer

Der Fully Connected Layer befindet sich am Ende des Convolutional Neural Networks und ist vom Aufbau identisch zu den in Fig. 5 gezeigten MLPs. Die Eingabe in diesen Layer ist ein eindimensionaler Tensor, jedoch geben die Convolutional bzw. Pooling Layer einen Tensor höherer Ordnung aus. Aus diesem Grund findet noch vor dem Fully Connected Layer eine flattening-Operation statt. Hierbei wird der Tensor dritter Ordnung in einen einfachen Vektor umgeformt. Ein solcher Tensor $X \in \mathbb{R}^{H \times W \times D}$ kann also in einen Vektor $b \in \mathbb{R}^N$ über (11)

$$b_k = A_{i,j,c} \tag{11}$$

mit $k = 1, 2, ..., N$ umgewandelt werden [12].

Der Vektor wird dem Fully Connected Layer übergeben. Über den in Sektion II erklärten Prozess führt dieser Layer eine Klassifizierung durch. Hierzu wird eine Matrixmultiplikation mit dem Vektor und den Gewichten des MLPs durchgeführt. Jedes Element im Ausgangsvektor ist eine gewichtete Summe der Eingaben, wobei die Gewichte während des Trainings gelernt werden. Die Ausgabe dieses Layers wird dann mit dem gewünschten Ergebnis verglichen und somit entsteht ein Trainingsprozess [7][11].

IV. Convolutional Autoencoder

Die gezeigten Modelle und Konzepte können nun zu einem Convolutional Autoencoder zusammengesetzt werden. Autoencoder verwenden unsupervised Learning und sind eine weit verbreite Art von künstlichen neuronalen Netzwerken. Sie über den Encoder in der Lage, einen Input mit hohen Dimensionen in einen Output niedrigerer Dimension zu transformieren. Im nächsten Schritt versucht der Autoencoder, aus dem kodierten Output den zu Beginn eingegebenen Input zu rekonstruieren. Wichtig ist hierbei, die Eingabedaten so genau wie möglich wiederherzustellen. Des Weiteren lernt der Autoencoder eine nützliche, kodierte Repräsenation der Daten, welche die wichtigsten Merkmale bzw. Features der Eingabe enthält.

Ihre Einsatzmöglichkeiten sind dabei sehr vielfältig. Aufgrund der Tatsache, dass diese die Daten stark komprimiert betrachten, sind sie besonders sinnvoll, wenn die Eingabedaten eine hohe Dimension besitzen und damit in einem gewöhnlichen MLP viel Speicherplatz und Laufzeit benötigen würden. Sie eignen sich ideal für das komprimieren von Videos und Bildern. Ebenfalls hat sich herausgestellt, dass sich diese sehr gut für die Rauschentfernung von Bilddateien eignen, da sie die Eingabedaten in eine komprimierte, weniger rauschempfindliche Repräsentation umwandeln können. Da Convolutional Autoencoder den bereits erläuterten Prozess der Konvolution nutzen, eignen diese sich noch besser für das Denoising von Bilddateien. Sie erkennen mit den Kerneln wichtige Muster in den Bildern und können so ein detailreiches Resultat erzeugen.

A. Architektur

In Fig. 9 ist die Architektur eines Autoencoders zu sehen. Im Allgemeinen besteht dieser aus zwei Teilen, einem Encoder und einem Decoder. Der Encoder ist

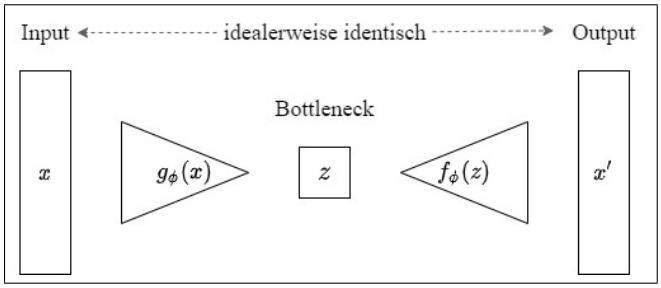

Fig. 9. Architektur eines Autoencoder

ein Convolutional Neural Network und transformiert den hoch-dimensionierten, originalen Input zu einem niedrigdimensionierten Zwischenergebnis. Der Decoder ist quasi ein CNN in umgekehrter Reihenfolge und versucht, die Daten von dieser Repräsentation wiederherzustellen. Hierzu wird das in iiierläuterte Upsampling verwendet. Wird ein Autoencoder für Rauschentfernung gebaut, so ist die Anzahl der Input Layer im Encoder und der Output Layer am Ende des Decoders identisch. Grund hierfür ist, dass die Ausgabe des Netzwerks nur eine verbesserte Darstellung der Eingabedaten sein und diese nicht räumlich vergrößern oder verkleinern soll. Für Anwendungen wie beispielsweise die Datensynthese wären aber auch unterschidliche Anzahlen an In- und Outputschichten denkbar. Mathematisch ausgedrückt wird ein Input x über die Enkodierfunktion g zu einer kodierten Repräsentation z über (12) umgewandelt.

$$\mathbf{z} = g_\phi(\mathbf{x}) \tag{12}$$

Dieser enkodierte Code z wird dann mit der Dekodierfunktion f über (13) zu einer möglichst verlustfreien Version der Eingabe mit

$$\mathbf{x}' = f_\theta(z) = f_\theta(g_\phi(\mathbf{x})) \tag{13}$$

transformiert. Die Parameter (θ, ϕ) werden über die Backpropagation trainiert und erzeugen so eine immer besser werdende, sich an die Originaleingabedaten annährende, Ausgabe [13].

$$\mathbf{x} \approx f_\theta(g_\phi(\mathbf{x})) = x' \tag{14}$$

B. Training

Wie jedes andere künstliche neuronale Netzwerk muss auch der Autoencoder trainiert werden, um zuverlässige Ergebnisse zu erhalten. Im ersten Schritt werden hierzu Trainingsdaten benötigt. In vielen Fällen ist die Menge der Trainingsdaten ein Problem, denn diese muss hinreichend groß sein. Bei einem Autoencoder für Rauschentfernung lassen sich jedoch leicht Trainingsdaten über das Hinzufügen von Rauschfiltern auf Bilder erzeugen. Eine weitere Möglichkeit besteht darin, bereits vorhandene, öffentliche Trainingsdatensätze wie den MNIST-Datensatz zu verwenden. Dem Autoencoder werden nun also zwei Datensätze zur Verfügung gestellt. Die verrauschten Bilder werden in den Autoencoder eingegeben und dieser versucht, eine rauschfreie Version des Bildes zu erzeugen. Das Ergebnis wird mit dem korrespondierenden rauschfreien Bild im Trainingsdatensatz verglichen und die Backpropagation startet, um die Gewichte und Biases weiter zu optimieren. Das Training wird beendet, wenn der Autoencoder keine Verbesserungen bzw. Aktualisierungen der Gewichte durchführt, das Ergebnis gar schlechter oder eine vorweg definierte Iterationsgrenze (Epoche) überschritten wird. [14]

V. Implementation

A. Aufbau

Im Folgenden wird ein einfacher Convolutional Autoencoder For Image Denoising implementiert. Hierfür

wird Python mit dem Keras Framework verwendet. In diesem Beispiel wird der MNIST Datensatz verwendet, welcher aus $28x28px$ Bildern von handgeschriebenen Zahlen besteht. Dem Datensatz wird ein künstliches gaußsches Rauschen hinzugefügt, um ein reales Szenario zu erzeugen.

B. Code

In dieser Arbeit werden lediglich die wichtigsten Elemente des Autoencoders anhand von Codeausschnitten erläutert, der vollständige Code ist verfügbar unter: https://github.com/tobiwnrt/Autoencoder_Image_Denoising

Zu Beginn wird der MNIST Datensatz, bestehend aus Trainings- und Testdaten, geladen und in zwei Variablen abgespeichert. Der zweite Wert des Tupels sind Label für die Daten, welche nicht benötigt und deshalb nicht gespeichert werden.

```
(x_train, _), (x_test, _) = mnist.load_data()
```

Diese Zeilen wandeln die Pixelwerte der beiden Datensätze von Integern in Floats um und normalisieren diese auf Werte zwischen 0-1, sodass diese ähnlich wie die Zahl Acht in Fig. 7 als Tensor dritter Ordnung repräsentiert werden können.

```
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
```

Nun wird den Datensätzen ein gaußsches Rauschen hinzugefügt. Der exemplarische Code für den Trainingsdatensatz lautet:

```
noise_factor = 0.5
x_train_noisy = x_train + noise_factor
    * np.random.
    normal(loc=0.0, scale=1.0,
    size=x_train.shape)
x_train_noisy = np.clip(x_train_noisy, 0., 1.)
```

Daraufhin wird der eigentliche Autoencoder aufgebaut. Für den Encoder werden zwei Konvolutions- und Poolingschichten hintereinandergereiht. Hier wird die ReLU als Aktivierungsfunktion verwendet.

```
x = Conv2D(32, (3, 3), activation='relu',
    padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(32, (3, 3), activation='relu',
    padding='same')(x)
encoded = MaxPooling2D((2, 2),
    padding='same')(x)
```

Ein den Dimensionen des Outputs des Encoders entsprechender Decoder wird nun aufgebaut. Er erzeugt ein Bild mit den Dimensionen der Eingabedaten. Die Sigmoidfunktion dient in diesem Fall als Aktivierungsfunktion.

```
x = Conv2D(32, (3, 3), activation='relu',
    padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(32, (3, 3), activation='relu',
    padding='same')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3),
    activation='sigmoid', padding='same')(x)
```

Im Anschluss wird der Trainingsprozess des Autoencoders gestartet. Hierbei wird eine Batchsize von 128 Bildern und 100 Epochen verwendet. Das bedeutet, dass in jeder Iteration 128 Bilder durch das Netz fließen und der gesamte Trainingsdatensatz 100-mal durchiteriert wird. Das Trainieren des Netzwerkes ist ein sehr aufweniger Prozess. In diesem Fall betrug die Trainingszeit mit einem Ryzen 5600 Prozessor 30 Minuten, mit einer NVIDIA RTX 3080 GPU und der CUDA-Technologie lediglich 10 Minuten.

```
autoencoder.fit(x_train_noisy, x_train,
    epochs=100,
    batch_size=128,
    shuffle=True,
    validation_data=(x_test_noisy, x_test),
    callbacks=[TensorBoard(log_dir='/tmp/tb',
    histogram_freq=0, write_graph=False)])
```

Im letzten Schritt werden die Testdaten in den Autoencoder eingegeben. Aus diesen wird nun mit den gelernten Mustern eine rauschfreie Version erstellt. Wie in Fig. 1 zu sehen ist, werden diese dann über das Framework Matplotlib zusammen mit den verrauschten Eingabedaten angezeigt.

C. Analyse

Derzeit existieren keine mathematischen Modelle, um die visuelle Analyse komplett zu formalisieren [1], jedoch ist in Fig. 1 zu erkennen wie der Autoencoder das Rauschen deutlich reduziert, wenn nicht gar vollständig eliminiert hat. Hierbei wurden die wichtigsten Features, also die Ecken und Kanten der Zahlen, beibehalten und nur minimal verändert.

VI. Fazit

Es wurde gezeigt, dass sich mit einfacher Mathematik die Grundlagen von Convolutional Autoencodern erklären lassen und diese exzellent geeignet sind, um Bilder zu entrauschen. Mit Hilfe von Frameworks wie Kera ist es selbst für einen Anfänger möglich, ein solches Netzwerk aufzubauen und zu verwenden. Die Forschungslage zu diesem Thema ist sehr gut und es werden stetig Verbesserungen und Erweiterungen für diese Art von neuronalem Netz vorgestellt. Durch die flexible Größe der Eingabedaten und ihre besondere Eigenschaft, Muster in Bildern und Videos erkennen zu können, existieren viele Einsatzgebiete für CAEs. Insbesondere ist hier die Medizintechnik zu nennen, welche Dank dieser Technologie schärfere Bildaufnahmen von MRI-, CT- und PET-Scans generieren und somit Leben retten kann.

REFERENZEN

[1] L. Fan et al., "Brief review of image denoising techniques", 8. Juli 2019. [online]. Verfügbar unter: https://vciba.springeropen.com/articles/10.1186/s42492-019-0016-7. [Letzter Zugriff am 10. März 2023].

[2] G. Abdul, S. Usman, "Convolutional Auto Encoder For Image Denoising", 31. Dez. 2021. [online]. Verfügbar unter: https://journals.umt.edu.pk/index.php/UMT-AIR/article/view/2467/924. [Letzter Zugriff am 10. März 2023].

[3] E. Grossia, M. Buscemab, "Introduction to artificial neural networks", Dezember 2007. [online]. Verfügbar unter: https://vciba.springeropen.com/articles/10.1186/s42492-019-0016-7. [Letzter Zugriff am 10. März 2023].

[4] Kukreja1 et al., "AN INTRODUCTION TO ARTIFICIAL NEURAL NETWORK", 2016. [online]. Verfügbar unter: https://www.researchgate.net/publication/319903816_AN_INTRODUCTION_TO_ARTIFICIAL_NEURAL_NETWORK. [Letzter Zugriff am 10. März 2023].

[5] W. Ertel, "Grundkurs Künstliche Intelligenz: Eine praxisorientierte Einführung", 5. Aufl. (Lehrbuch), Wiesbaden: Springer Vieweg, 2021.

[6] A. Faulmann et al., "NEURONALE NETZE", 1. Mai 2017. [online]. Verfügbar unter: https://www.mmf.univie.ac.at/fileadmin/user_upload/p_mathematikmachtfreunde/Materialien/MmF_NeuronNetze_Faulman-Lackinger-Prenner-Resch_1.0.pdf. [Letzter Zugriff am 10. März 2023].

[7] J. Wu, "Introduction to Convolutional Neural Networks", 1. Mai 2017. [online]. Verfügbar unter: https://cs.nju.edu.cn/wujx/paper/CNN.pdf. [Letzter Zugriff am 10. März 2023].

[8] A. Krizhevsky et al., "ImageNet Classification with Deep Convolutional Neural Networks", Jan. 2012. [online]. Verfügbar unter: https://cs.nju.edu.cn/wujx/paper/CNN.pdf. [Letzter Zugriff am 10. März 2023].

[9] K. He, J. Sun, "Convolutional Neural Networks at Constrained Time Cost", 4. Dez. 2014. [online]. Verfügbar unter: https://arxiv.org/pdf/1412.1710.pdf. [Letzter Zugriff am 10. März 2023].

[10] H. Gholamalinezhad, H. Khosravi, "Pooling Methods in Deep Neural Networks, a Review", Sept. 2020. [online]. Verfügbar unter: https://arxiv.org/ftp/arxiv/papers/2009/2009.07485.pdf. [Letzter Zugriff am 10. März 2023].

[11] S. Albawi, T. Mohammed, S. Al-Zawi, "Understanding of a Convolutional Neural Network", 21. Aug. 2017. [online]. Verfügbar unter: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8308186. [Letzter Zugriff am 10. März 2023].

[12] H. Leea, J. Song, "Introduction to convolutional neural network using Keras;an understanding from a statistician", Nov. 2019. [online]. Verfügbar unter: https://koreascience.kr/article/JAKO201910861317398.pdf. [Letzter Zugriff am 10. März 2023].

[13] L. Gondara, "Medical image denoising using convolutional denoising autoencoders", 18. Sept. 2019. [online]. Verfügbar unter: https://arxiv.org/pdf/1608.04667.pdf. [Letzter Zugriff am 10. März 2023].

[14] P. Venkataraman, "Image Denoising Using Convolutional Autoencoder", 24. Jul. 2022 [online]. Verfügbar unter: https://arxiv.org/pdf/2207.11771.pdf. [Letzter Zugriff am 10. März 2023].