HAW HAMBURG

BACHELORTHESIS
Sofian Wüsthoff

3D-Tiefenbildrekonstruktion mit Hilfe von Faltungsnetzwerken

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Sofian Wüsthoff

3D-Tiefenbildrekonstruktion mit Hilfe von Faltungsnetzwerken

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Informatik Technischer Systeme*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg
Betreuender Prüfer: Prof. Dr.-Ing. Andreas Meisel
Zweitgutachter: Prof. Dr. Philipp Jenke
Eingereicht am: 30. März 2022

Sofian Wüsthoff

Thema der Arbeit
3D-Tiefenbildrekonstruktion mit Hilfe von Faltungsnetzwerken

Stichworte
Tiefenbilderkennung, Deep Learning, Bildverarbeitung, Neuronale Netze, Faltungsnetzwerke, Autoencoder

Kurzzusammenfassung
In dieser Bachelorarbeit werden Faltungsnetzwerke zur monokularen Tiefenbildrekonstruktion genutzt. Ziel dabei ist zu untersuchen, ob ein einfach gehaltenes Netz in der Lage ist, Tiefenbilder aus RGB-Bildern zu rekonstruieren und zu analysieren, nach welchen Prinzipien und Merkmalen dies stattfindet. Dazu wurden Szenarien erstellt, in denen verschiedene Hyperparameter des Netzes verändert wurden. Für jedes Szenario wird eine Vielfalt an Ergebnissen dargestellt, um daraus Erkenntnisse über die Auswirkung der Veränderungen zu gewinnen und Annahmen über die Vorgehensweisen beim Lernen der Tiefeninformation zu treffen.

Sofian Wüsthoff

Title of Thesis
3D Depth Reconstruction with the help of Convolutional Networks

Keywords
Depth estimation, deep learning, image processing, neural networks, convolutional neural networks, autoencoder

Abstract
In this thesis, convolutional neural networks are used for monocular depth reconstruction. The goal is to examine, whether it is possible for a simple network to reconstruct depth images from an RGB-image and to analyse, which principals and features are used to do so. Therefore, different scenarios were constructed, in which different hyperparameters of the network were modified. For each scenario, a variety of results are shown, to gain insights into the effects of the modifications and hypothesize methods used, to learn the depth information.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Die Künstliche Intelligenz erscheint oftmals als eine hochkomplexe Technologie, die noch weit entfernt in der Zukunft lieg, dabei ist diese bereits sehr stark verbreitet. Anwendungen wie Sprachübersetzer, Suchmaschinen, Sprachassistenten oder Computerspiele sind Teil der künstlichen Intelligenz und in so gut wie jedem Smartphone enthalten. Neben alltäglicher Nutzung werden jedoch ebenfalls Computersysteme entwickelt, die auf Augenhöhe mit dem Menschen Aufgaben erledigen.

Computer Vision beschreibt die Analyse aufgenommener Bilder, um daraus Inhalte oder geometrische Informationen zu entnehmen. Zusammen mit dem maschinellen Lernen und Neuronalen Netzen, wird dabei versucht das menschliche Sehen zu imitieren und darauf aufbauend Aufgaben zu lösen, die teilweise selbst für den Menschen zu schwierig sind. Dabei ist das Erkennen der Tiefen in einem Bild, einer der Herausforderungen. Die Tiefenbild-rekonstruktion, oder auch Tiefenvorhersage, versucht für jeden Pixel eines Bildes einen Tiefenwert zu bestimmen. Bisherige Ansätze nutzen bei der Tiefenbildrekonstruktion oftmals Stereoskopie. Die Stereoskopie nutzt dabei das Prinzip des binokularen Sehens, wie es beim Menschen bekannt ist. Der Mensch hat zwei Augen und kann daher seine Umgebung aus zwei Blickwinkeln betrachten, was ihm erlaubt eine bessere Wahrnehmung des Raumes zu haben und Entfernungen besser einschätzen zu können. Dies erreicht die Stereoskopie durch Nutzung von zwei Bildern, die aus verschiedenen Winkeln aufgenommen wurden. Der Gegensatz dazu ist das monokulare Sehen, bei dem nur ein Bild verwendet wird, so wie es von herkömmlichen Kameras bekannt ist. Dadurch existiert eine Reihe an Bildern, die für eine Tiefenbild-rekonstruktion genutzt werden könnten. Bereiche wie soziale Medien, Online-Shopping, Immobilienmarkt, oder autonomes Fahren stellen dabei einige Anwendungsmöglichkeiten dar. Die Tiefe aus einem einzelnen Bild zu entnehmen ist allerdings kein einfacher Prozess und benötigt einige Merkmalen, wie Perspektive, Objektgröße oder Bildposition. Daher wurde dieser Bereich der Tiefenbildrekonstruktion noch nicht im gleichen Maße untersucht.

1.1 Zielsetzung

Das grundlegende Ziel dieser Arbeit liegt in der Untersuchung der monokularen 3D-Tiefen-bildrekonstruktion. Dabei sind zwei Hauptziele definiert: Beurteilung, ob ein einfaches Neuronales Netz bereits in der Lage ist, diese komplexe Aufgabe zu erfüllen und die Untersuchung dessen, woran sich ein solches Neuronales Netz, bei dem Lernen der Tiefeninformationen der Bilder, richtet. Dazu wurde in einem ersten Schritt die passende

Architektur ausgewählt. Anschließend wurde diese anhand verschiedener Szenarien untersucht und angepasst, um ein zufriedenstellendes Ergebnis zu erreichen. Bei der Anpassungen wurden vereinzelt Hyperparameter angepasst, um deren Auswirkung auf das Training und das Ergebnis zu evaluieren. Aus den verschiedenen Ergebnissen und der Architektur wird zusätzlich untersucht, wie das Netz, aus der vorhandenen Datengrundlage, die Tiefe rekonstruieren kann.

1.2 Struktur der Arbeit

Diese Arbeit beinhaltet, inklusive der Einleitung, insgesamt 6 Kapitel.

- Kapitel 2 führt, durch die Grundlagen, weiter in die Arbeit ein und umfasst Hintergrundwissen zum Thema Machine Learning.
- Kapitel 3 stellt kurz relevante Arbeiten vor, die das Thema der Tiefenbildrekonstruktion behandeln.
- Kapitel 4 beinhaltet Informationen zur Arbeitsumgebung und Ressourcen, wie Hardware, Software und Datensatz, die verwendet wurden.
- Kapitel 5 beschreibt die Vorgehensweise dieser Arbeit und erläutert die expliziten Schritte. Zusätzlich werden die szenarienbasierten Untersuchungen vorgestellt und dessen Ergebnisse evaluiert.
- Kapitel 6 fasst die Arbeit zusammen. Es wird ein Fazit gezogen und ein Ausblick mit möglicher Weiterführung des Themas dieser Arbeit gegeben.

2 Grundlagen

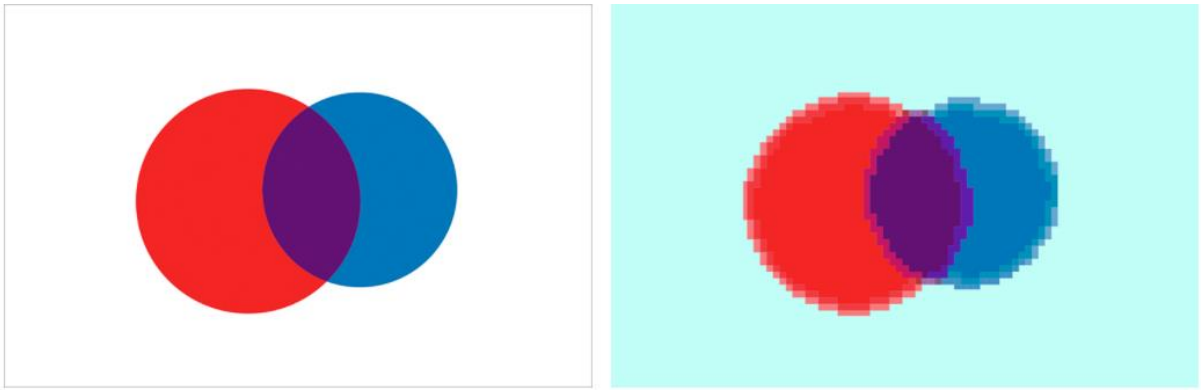
2.1 Digitale Bilder und Computer Vision

2.1.1 Raster- und Vektorgrafiken

Digitale Bilder werden durch Zahlen dargestellt und können somit von einem Computer verarbeitet werden. Dabei werden zwischen zwei Arten unterschieden, die Raster- oder Vektorgrafik.

Rastergrafiken bestehen aus einem Raster von Bildpunkten, auch genannt Pixel, die einer Koordinate zugeordnet sind und somit das Bild darstellen. Jedes dieser Pixel enthält einen Farbwert, der sich aus den drei Farbkanälen Rot, Grün und Blau ergibt. Zusammen ergeben diese den Farbraum RGB. Für jeden Farbkanal ist ein Wert hinterlegt, der 8-bit groß ist und die Intensität der jeweiligen Farbe festlegt. Somit liegt der Wert jedes Kanals zwischen 0 und 255 und gibt einem Pixel eine Größe von 24-bit.

Vektorgrafiken hingegen nutzen mathematische Beschreibungen verschiedener Objekte, die Informationen, wie die Position, Größe und Farbe enthalten. Dadurch haben Vektorgrafiken nicht nur eine kleinere Speichergröße, sondern lassen sich beliebig skalieren, ohne an Qualität zu verlieren.



2-1 Vergleich zwischen Vektorgrafik(l.) und Rastergrafik(r.) *(Adobe., Zugriff: 2022)*

Um eine Rastergrafik hochauflösend darstellen zu können, bedarf es an einer hohen Anzahl Pixel. Als heutiger Standard wird oft eine Bildgröße von 1920 x 1080 gesehen, was 1920 Pixel in der Breite und 1080 Pixel in der Höhe bedeutet und eine gesamte Pixelzahl von 2.073.600 ergibt. Rastergrafiken sind allerdings weiterhin von größter Wichtigkeit, da die reine Zahlenrepräsentation in vielen Anwendungen gebraucht wird.

2.1.2 Computer Vision

Computer Vision beschreibt die Analyse und Verarbeitung von Bildern, um daraus Informationen zu gewinnen. Dieser Bereich hängt stark von der Zahlenrepräsentation ab, um mathematische Operationen auf diesen ausführen zu können, die beispielsweise Bildverarbeitung, Merkmalsextraktion sowie Muster- und Objekterkennung ermöglichen.

Bildverarbeitung

Die Bildverarbeitung beschreibt verschiedene Operationen, die auf einem Bild angewendet werden können. Dazu gehören zum Beispiel affine Transformationen, um Bilder zu skalieren, drehen, spiegeln oder zu verschieben, oder Tief- und Hochpässe, um ein Bild zu filtern, die Schärfe zu verändern und Kanten hervorzuheben.

Merkmalsextraktion und Mustererkennung

Merkmalsextraktion und Mustererkennung nutzen die Bildverarbeitung, um gewisse Merkmale oder Muster in einem Bild zu erkennen. Beliebte Beispiele sind hierbei die Kantendetektion, um geometrische Formen oder Umrisse zu entnehmen, und die Bildsegmentierung, bei der Bildbereiche gruppiert werden, wenn sie einen Zusammenhang aufweisen.

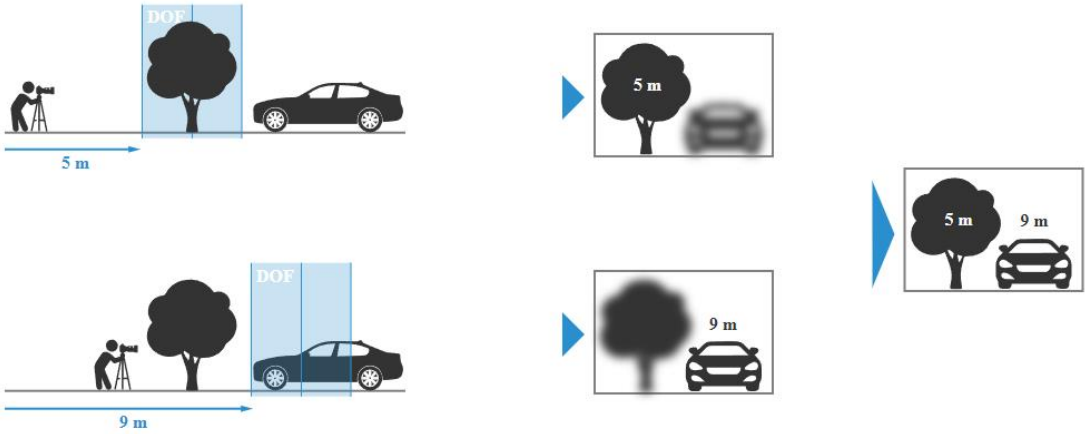
Objekterkennung

Als Erweiterung und Kombination der Merkmalsextraktion und Mustererkennung, wird die Objekterkennung bezeichnet. Aus den erkannten Merkmalen und Mustern werden Verknüpfungen zu gewissen Objekten gemacht. Diese Aufgabe ist deutlich komplexer und bedarf daher schon an Deep Learning.

Shape from X

All die zuvor erwähnten Verfahren basieren auf Zweidimensionalität, somit wäre der nächste Schritt der Übergang in das Dreidimensionale. Dabei stellt sich die Aufgabe herauszufinden, welche Eigenschaften eines 2D-Bildes, die 3D-Form verschiedener Objekte oder ganzer Bilder andeuten und wie. Shape from X behandelt dabei verschiedene Ansätze, die diverse Eigenschaften, wie den Fokus, die Schattierung, Silhouette und Textur eines Bildes nutzen, um die 3D-Formen daraus abzuleiten.

Shape from (de)-focus, oder auch Fokusvariation, nutzt verschiedene Fokusstufen einer Kamera, um daraus die Tiefe von Objekten und somit des Bildes zu bestimmen. Bekannterweise kann beim Fotografieren die Schärfe nur auf den Vorder- oder Hintergrund eingestellt werden. Es wird also jeweils ein Foto gemacht, indem entweder der Vorder- oder Hintergrund scharf ist. Mathematisch wird dann bestimmt, welche Bildpunkte am schärfsten sind und zusammen mit der Information, auf welcher Entfernung das Foto aufgenommen wurde, kann die Tiefe der jeweiligen Teile des Bildes bestimmt werden.

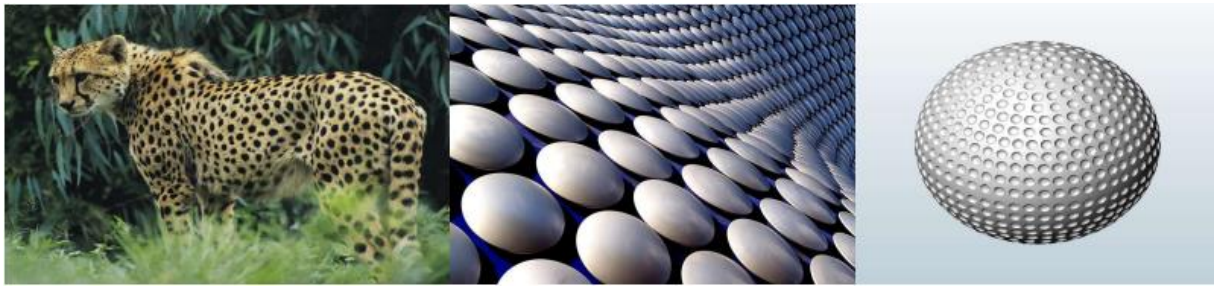


2-2 Beispiel für die Funktion Fokusvariation *(Bruker., Zugriff: 2022)*

Der Shape from Shading Ansatz nutzt mehrere Fotos eines Objektes, auf denen dieses aus verschiedenen Richtungen beleuchtet wird. Die Relation zwischen der Reflexion des auftreffenden Lichtes auf das Objekt und das von der Kamera aufgenommene Licht werden genutzt, um die Oberfläche, und somit auch Form des Objektes, zu ermitteln.

Beim Shape from Silhouette oder auch Silhouetten-Schnittverfahren werden mehrere Bilder, aus verschiedenen Perspektiven, genutzt, um ein Objekt von seinem Hintergrund zu trennen und aus dem Bild zu „schneiden“. Dabei muss die Position, Ausrichtung und Einstellung der Kamera beachtet werden. Aus den verschiedenen Perspektiven kann dann die jeweilige Silhouette des Objektes entnommen werden, woraus die Form ermittelt werden kann.

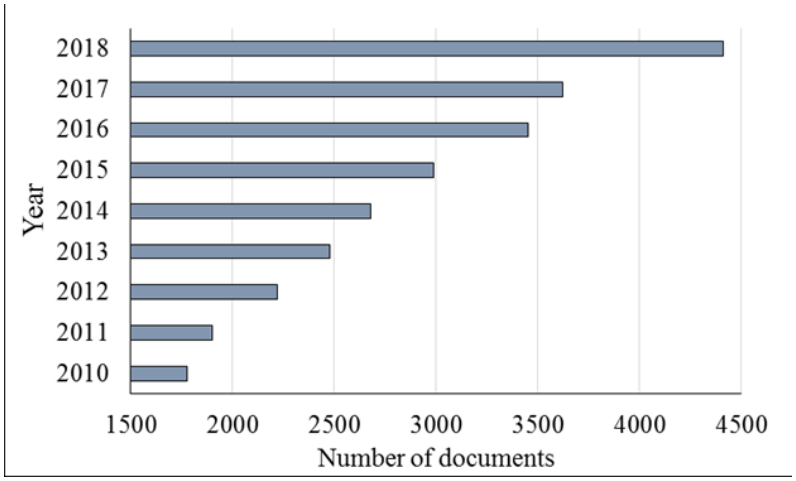
Shape from Texture nutzt bekannte Muster einer Textur und dessen Veränderung, um die 3D-Form eines Objektes zu bestimmen, auch Texturgradient genannt, (Krantz, Zugriff: 2022). Wird beispielsweise eine Fläche mit mehreren Kreisen der gleichen Größe, als Textur, verbogen, verzerren sich einige der Kreise oder werden kleiner. Daraus kann abgeleitet werden, dass sich die Form des Objektes, in dem Fall die Fläche, verändert hat. Dabei kommt die zuvor erwähnte Mustererkennung ins Spiel, aus der ein Muster, hier die gleichgroßen Kreise, erkannt wird und die Veränderung der Kreise auf die Form deuten.



2-3 Beispiele von verzerrten Texturen *(Sourtzinos, Zugriff: 2022)*

2.2 Machine Learning

Machine Learning ist ein Unterbegriff der Künstlichen Intelligenz und hat in den letzten Jahren einen starken Aufwärtstrend erlebt.



2-4 Anzahl der Dokumente zu Machine Learning von 2010-2018 *(Ardabili, et al., 2019)*

Die Theorie hinter Machine Learning exisitiert schon seit vielen Jahren, was an Veröffentlichungen, wie (Rosenblatt, 1958) oder (Solomonoff, 1957) zu sehen ist. Bereits in diesen frühen Ansätzen wurde realisiert, dass das menschliche Denken, zu einem gewissen Grad, von Computern repliziert werden kann. Der Grund dafür, dass der Trend erst in den letzten 10 Jahren rapide angestiegen ist, ist die Entwicklung der Hardwareleistung, besonders bei Grafikkarten. Da Prozessoren, kurz CPUs, auf Schnelligkeit optimiert wurden, ist es ihnen möglich, kleinere Datenmengen schneller abzufragen. Graffikkarten, kurz GPUs, sind zwar langsamer, allerdings in der Lage eine höhere Bandbreite an Daten zu übertragen. Diese ist bei zahlreichen Machine Learning Anwendungen nötig, da mehrdimensionale Matrixmultiplikationen durchgeführt werden.

Machine Learning nutzt die Eingabe ausgewählter Daten und einer erwünschten Ausgabe, auch Label genannt, die in ein System gegeben werden, woraus dieses gewisse Regeln lernen soll, um die Eingabe in die Ausgabe umzuwandeln. Diese Regeln entstehen während des Lernens und sollen dem System erlauben die Regeln auf jeglichen Daten anzuwenden, um die gewünschte Ausgabe zu erhalten.

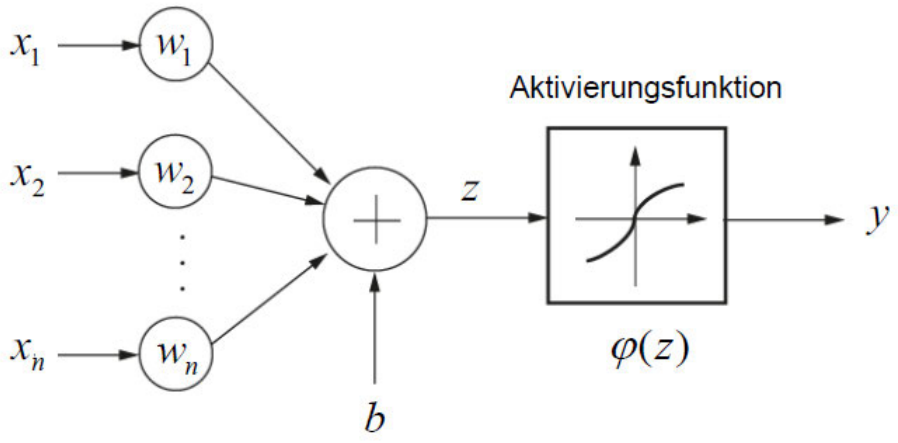
Deep Learning ist ein Teilbereich von Machine Learning und erweitert diesen dadurch, dass der Computer alle Aufgaben übernimmt. Dazu werden große Datenmengen und neuronale Netze genutzt, um die Aufgaben zu lösen. Der Mensch spielt dabei nur noch die Rolle, die Daten bereitzustellen. Klassische Beispiele hierbei sind die Objekterkennung, Bildklassifizierung oder Spracherkennung.

2.3 Künstliche Neuronale Netze

Künstliche neuronale Netze, oder auch nur neuronale Netze genannt, sind die Basis, auf der die Machine Learning Modelle aufbauen. Bestehend aus mehreren künstlichen Neuronen, soll dies die Funktionsweise des menschlichen Gehirns, auf sehr vereinfachter Weise, imitieren.

2.3.1 Neuron

Ein künstliches Neuron besteht aus einem Eingang, Bias und Gewichten, welche zusammen durch eine Aktivierungsfunktion geleitet und als Ausgang ausgegeben werden.



2-5 Aufbau eines Neurons *(Meisel, 2021)*

In Abbildung *2-5* ist das Zusammenspiel dieser einzelnen Komponenten dargestellt. Die Eingänge x_{1-n} werden jeweils mit einem Gewicht w_{1-n} multipliziert und aufsummiert. Anschließend wird ein Bias b addiert und das Ergebnis wird als Eingabe z in die Aktivierungsfunktion $\varphi(z)$ gegeben, aus der y als Ausgang entsteht.

2.3.2 Aktivierungsfunktion

Die Aktivierungsfunktion ist ein wichtiger Teil eines Neurons und kann in verschiedenen Varianten angewendet werden. Die Funktionen können linear, nicht-linear, stückweise linear oder eine Sprungfunktion sein. Eine Gemeinsamkeit verbindet diese allerdings, sie sind monoton steigend. Das heißt der Funktionswert bleibt immer gleich oder er steigt.

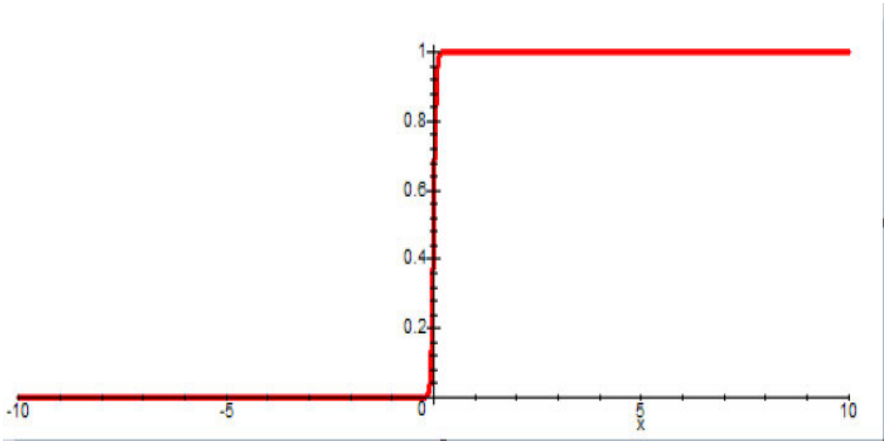
Im Folgenden werden kurz einige Aktivierungsfunktionen und ihre Funktionsweise erläutert.
Die Differenzierbarkeit bei diesen Funktionen ist wichtig, da dies bei der gradienten-basierten Optimierung vorausgesetzt wird.

Schwellenwertfunktion/Schrittfunktion

Die Schwellenwertfunktion gibt für jeden Wert kleiner als 0 eine 0 zurück und für jeden Wert größer als 0 eine 1. Diese Funktion hat einen großen Nachteil, da sie nicht differenzierbar ist und wird daher selten verwendet.
Formel:

$$\varphi^{hlim}(z) = \begin{cases} 0: z < 0 \\ 1: z \geq 0 \end{cases} \quad (1)$$

Graphische Darstellung:



2-6 Schwellenwertfunktion in graphischer Darstellung *(Meisel, 2021)*

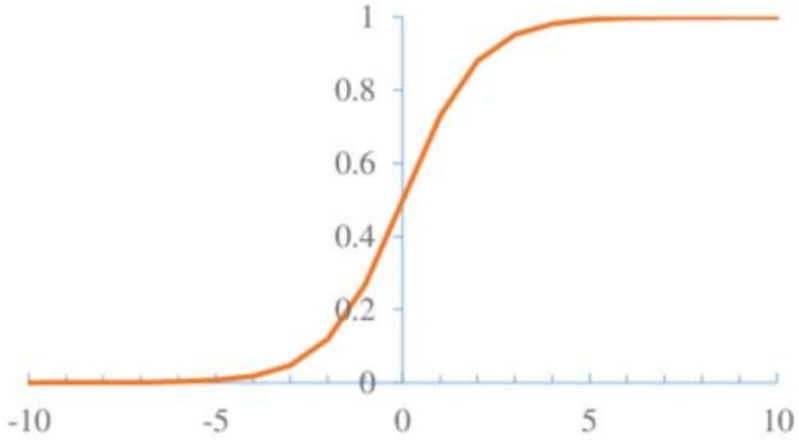
Sigmoid/Logistische Funktion

Sigmoid wird, im Gegensatz zu der Schwellenwertfunktion, häufiger verwendet, da die Werte in einem Intervall zwischen 0 und 1 liegen. Dies macht sie somit differenzierbar.

Formel:

$$\varphi^{sig}(z) = \frac{1}{1+e^{-z}} \tag{2}$$

Graphische Darstellung:



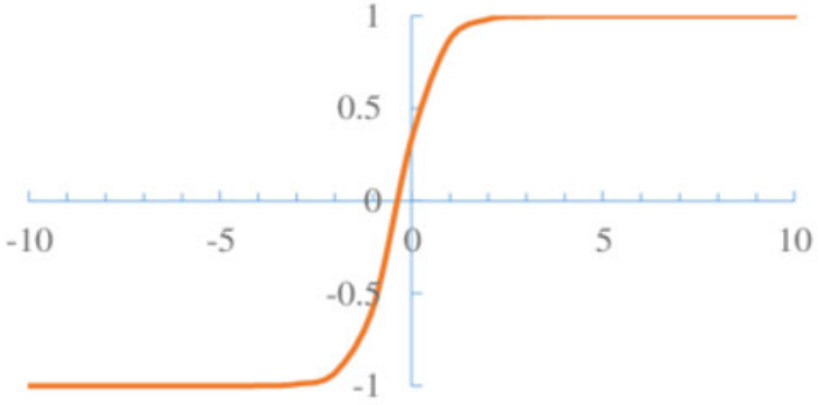
2-7 Sigmoid in graphischer Darstellung *(Jing, et al., 2018)*

Tangens hyperbolicus (tanh)

Auch diese Funktion ist differenzierbar, da dessen Wertebereich zwischen -1 und 1 liegt.
Formel:

$$\varphi^{tanh}(z) = tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

Graphische Darstellung:



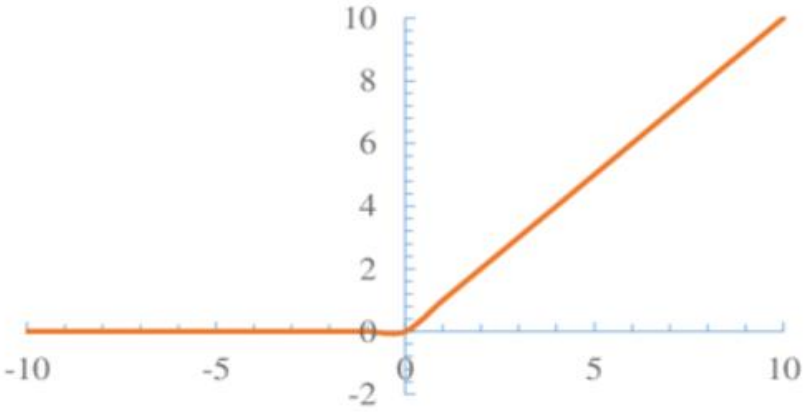
2-8 Tanh in graphischer Darstellung *(Jing, et al., 2018)*

Rectifier linear unit (ReLU)

Die ReLU Funktion hat den Wertebereich 0 bis unendlich und verläuft linear. Dadurch ist auch diese Funktion differenzierbar.
Formel:

$$\varphi(z) = max(0, z) \tag{3}$$

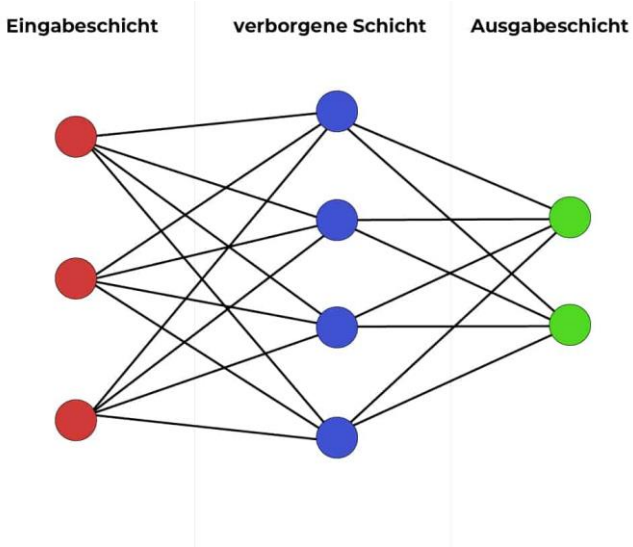
Graphische Darstellung:



2-9 ReLU in graphischer Darstellung *(Jing, et al., 2018)*

2.3.3 Aufbau des Neuronales Netzes

Das Verwenden von mehreren Neuronen und die Anordnung in einer gewissen Struktur, wird als Neuronales Netz bezeichnet. Dabei gibt es unterschiedliche Arten im Aussehen der Struktur, jedoch haben diese immer eine gemeinsame Grundstruktur, die in Abbildung *2-10* zu sehen ist.



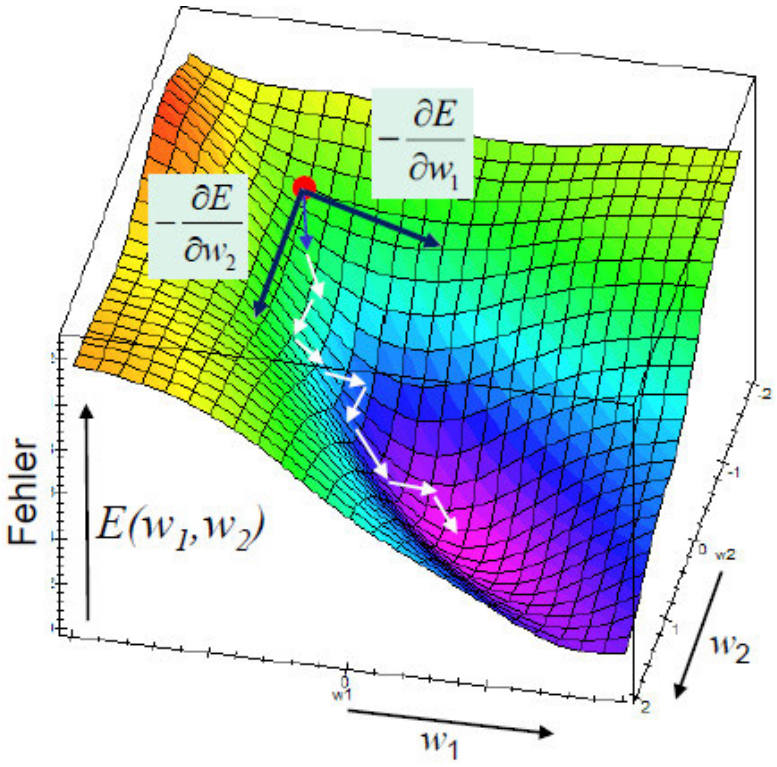
2-10 Grundstruktur eines NNs *(Wuttke, Zugriff: 2022)*

Die Eingabeschicht, oder Input Layer, besteht aus mehreren Eingaben, wie zum Beispiel mehrere Bildpunkte oder Signale. Die verborgene Schicht, auch Hidden Layer, beinhaltet ein oder mehrere Neuronen. In diesem Punkt unterscheiden sich die verschiedenen Netze voneinander, da je nach Netz unterschiedlich viele verborgene Schichten genutzt werden. Am Ende befinden sich die Ausgabeschicht, oder Output Layer, welche ebenfalls aus Neuronen besteht. Jedes der Neuronen im Netz hat den Aufbau aus Abbildung *2-5*, führt Berechnungen auf dessen Eingang aus und leitet das Ergebnis an die nächste Schicht weiter.

2.3.4 Training des Neuronalen Netzes

Bekannterweise findet in einem neuronalen Netz ein Training statt. Allgemein bedeutet Training, dass eine Verbesserung angestrebt wird, welche in der Praxis beurteilt wird. In Bezug auf neuronale Netze wird die Beurteilung von einer Loss Funktion (deutsch: Fehler- oder Kostenfunktion) übernommen. Grob zusammengefasst wird dabei die Ausgabe des Netztes mit der gewünschten Ausgabe „verglichen“. Oftmals wird dort die Differenz zwischen gewünschter und tatsächlicher Ausgabe als Richtwert genommen. Aus dem daraus berechneten Fehler sollen die Gewichte angepasst werden, um näher an die gewünschte Ausgabe zu kommen. Dazu wird der Gradientenabstieg genutzt.

Gradientenabstieg



2-11 Gradientenabstieg in einem Fehlergebirge *(Meisel, 2021)*

In Abbildung *2-11* wird der Gradientenabstieg, als Fehlergebirge visualisiert, dargestellt. Ziel hierbei ist es, von hohen Loss Werten auf niedrige herabzusteigen und dabei den steilsten Abstieg zu nehmen. Dieser kann aus dem negativen Gradienten berechnet werden. In der Abbildung wird von zwei Neuronen ausgegangen und somit von den zwei Gewichten $w1$ und $w2$. Der Gradient wird aus

$$\vec{\nabla} E(w_1, w_2) = \begin{pmatrix} \frac{\partial E}{\partial w_1} \\ \frac{\partial E}{\partial w_2} \end{pmatrix} \tag{4}$$

berechnet. Mit $\vec{\nabla}E$ kann durch

$$\vec{w}_{n+1} = \vec{w}_n - \eta \cdot \vec{\nabla}E \tag{5}$$

das Gewicht aktualisiert werden, sodass der Fehler kleiner wird. η ist hierbei der Schrittweitenfaktor und beeinflusst, wie weit der jeweilige Schritt des Gradientenabstiegs, in die entsprechende Richtung, sein soll.

Backpropagation

Die Backpropagation beschreibt eine Rückwärtsberechnung, die unter Nutzung der Zwischenergebnisse des Loss, der Ausgabe und der Multiplikation der Gewichte, den Gradienten bestimmen kann. Diese Zwischenergebnisse ergeben sich aus der forward propagation (deutsch: Vorwärtsberechnung).

Schritte der forward propagation am Beispiel von Zwei Eingabepunkten:

1. Multiplikation der Eingabepunkte mit den Gewichten und Summierung
$$z = x_1 w_1 + x_2 w_2 \tag{6}$$
2. Aktivierungsfunktion
$$y = \varphi(z) \tag{7}$$
3. Loss Funktion z.B.
$$E = \frac{1}{2} \cdot (t - y)^2 \tag{8}$$

Aus diesen Schritten kann dann mithilfe der partiellen Ableitung der einzelnen Funktionen, der Gradient berechnet werden, was den Backpropagation-Teil konstituiert:

1. Ableitung von E nach y
$$\frac{\partial E}{\partial y} = (y - t) \tag{9}$$
2. Ableitung von y nach z
$$\frac{\partial y}{\partial z} = \varphi'(z) \tag{10}$$
3. Ableitung von z nach w_i
$$\frac{\partial z}{w_i} = x_i \tag{11}$$
4. Multiplikation von 1. – 3. Ergibt Gradient: $\frac{\partial E}{w_i}$

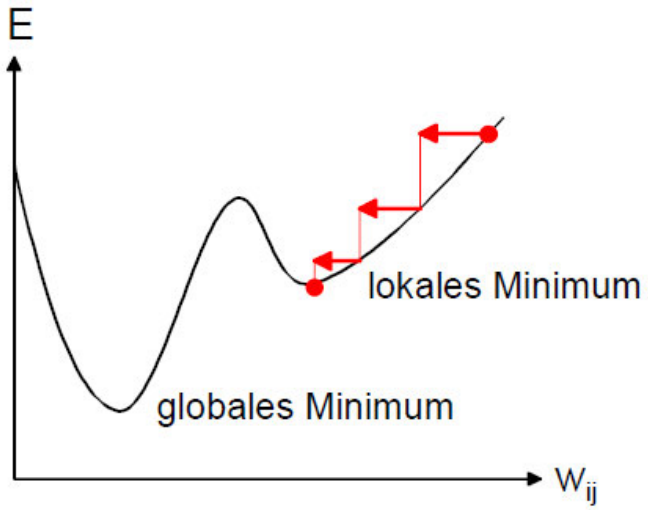
Dieser Durchlauf wird für w_1 und w_2 durchgeführt, woraus sich der Gradient $\vec{\nabla}E$ ergibt.

2.3.5 Probleme beim Lernen

Backpropagation Probleme

Der Backpropagation-Algorithmus zieht einige potenzielle Probleme mit sich, wie lokale Minima, Plateaus, Oszillation in steilen Bereichen oder Aussprung aus steilen Minima in der Fehlerfunktion. Im Folgenden werden diese, zusammen mit möglichen Lösungen, beschrieben.

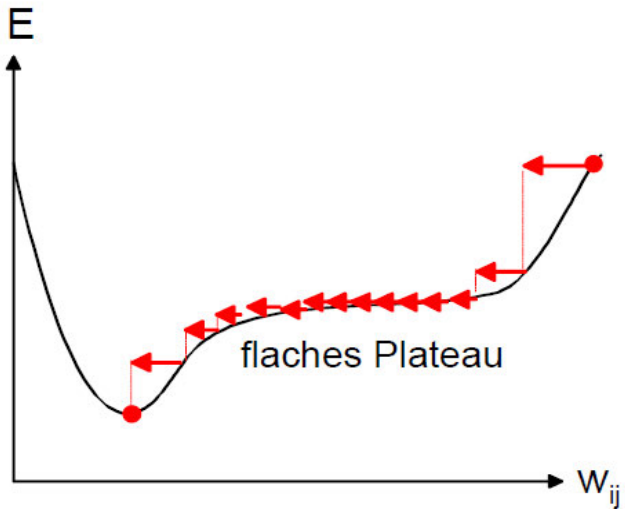
Lokale Minima in der Fehlerfunktion



2-12 Lokale Minima in der Fehlerfunktion *(Meisel, 2021)*

Die Schrittweite des Gradientenabstiegs ist abhängig aus dem konstanten, in Abschnitt *2.3.4* erwähnten, Schrittweitenfaktor und dem lokalen Gradienten. Wird, wie in Abbildung *2-12*, ein lokales Minimum erreicht, ist der lokale Gradient sehr gering und somit auch die Schrittweite. Dadurch wird das Training sehr langsam oder stopp sogar komplett.

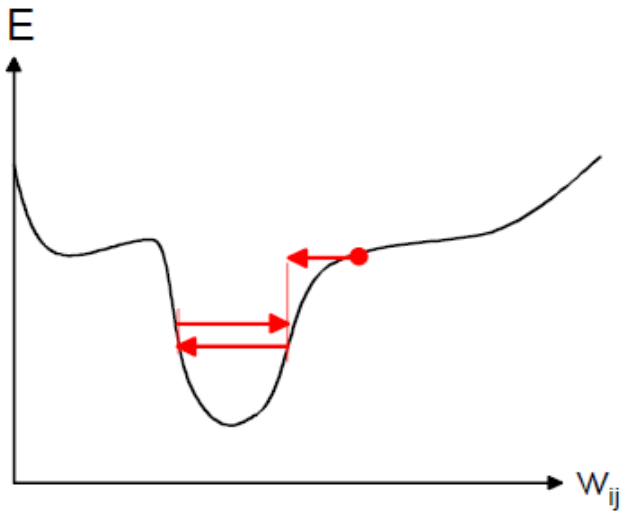
Plateaus in der Fehlerfunktion



2-13 Plateau in der Fehlerfunktion *(Meisel, 2021)*

Durch den flachen Bereich in der Funktion, ist der Gradient gering und somit auch die Schrittweite. Dies führt zu einem sehr langsamen Training.

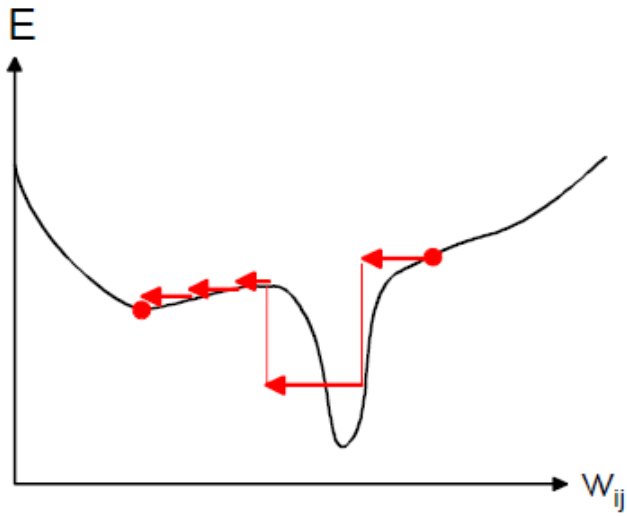
Oszillation in steilen Bereichen



2-14 Oszillation in der Fehlerfunktion *(Meisel, 2021)*

In einem steilen Bereich der Fehlerfunktion kann es zu einem starken Gradientenabstieg kommen. In der, in Abbildung *2-14* zu sehenden, „Schlucht“ versucht der Backpropagation-Algorithmus weiterhin den steilsten Gradientenabstieg zu finden. Dies führt dazu, dass dieser immer wieder nach unten steigt und in der „Schlucht“ stecken bleibt.

Aussprung aus steilen Minima

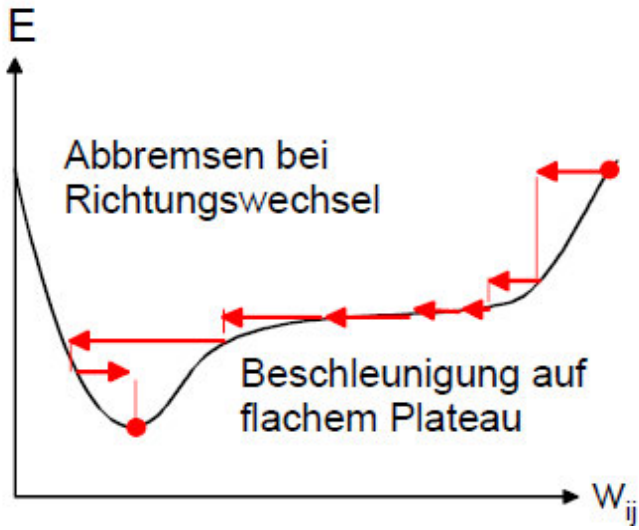


2-15 Aussprung aus einem Minimum in der Fehlerfunktion *(Meisel, 2021)*

Eine recht hohe Schrittweite allerdings kann allerdings dazu führen, dass der Algorithmus aus einem Minimum wieder herausspringt und der Fehler deutlich ansteigt.

Momentum Term

Eine der möglichen Lösungen, der Momentum Term, berücksichtigt gewichtet die vorherige Schrittweite und wird bei der Berechnung der nächsten Gewichte dazu addiert. Bei flachen Teilen der Fehlerfunktion wird mit jedem Schritt die Schrittweite erhöht, was das Training verschnellert. Bei einem Minimum wird nicht so schnell wieder ausgestiegen oder oszilliert, da die Schrittweite abgebremst wird. Das geschieht durch den Richtungswechsel und der vorherigen Schrittweite, die noch in die andere Richtung zeigt.



2-16 Wirkung des Momentum Terms in der Fehlerfunktion *(Meisel, 2021)*

Resilient Propagation (Rprop)

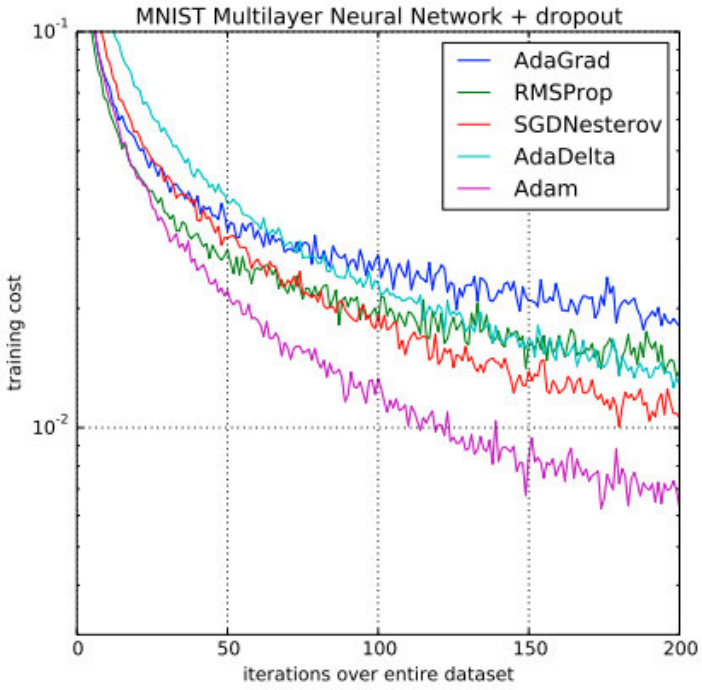
Resilient Propagation ermöglicht adaptive Schrittweitenfaktoren, die sich abhängig vom Gewicht berechnen. Dadurch wird der Momentum Term durch ein dynamisches Verfahren ausgetauscht. Wenn sich die Gradientenrichtung über mehrere Schritte nicht verändert, wird der Schrittweitenfaktor erhöht und wenn sich die Richtung verändert, wird dieser kleiner. Somit wird die Schrittweite unabhängig vom Betrag und nur von der Richtung des Gradienten verändert. Dadurch werden langsames Training auf Plateaus, Erreichen von lokalen Minima, Oszillation und Aussprung aus Minima verhindert.

Optimizer

Optimizer (deutsch: Optimierer) sind weitere adaptive Verfahren, die das Training verbessern und sich an die Gewichte anpassen. Dabei gibt es eine Reihe an verschiedenen Optimierern. Einige nennenswerte werden im Folgenden kurz beschrieben:

- **Adaptive Gradient Descent (AdaGrad)**: Nutzt das Quadrat der vorherigen Schrittweiten, wodurch schneller ein Tiefpunkt erreicht werden kann
- **AdaDelta**: Erweiterung von AdaGrad. Versucht das aggressive und monotone reduzieren der Schrittweite von AdaGrad zu vermeiden
- **Adaptive Moment Estimation (Adam)**: Nutzt einen fortlaufenden, exponentiellen Durchschnitt der vorherigen Schrittweiten und hat sich als den effektivsten Optimizer herausgestellt (Kingma, et al., 2017)

In Abbildung *2-17* ist ein Vergleich verschiedener Verfahren dargestellt.

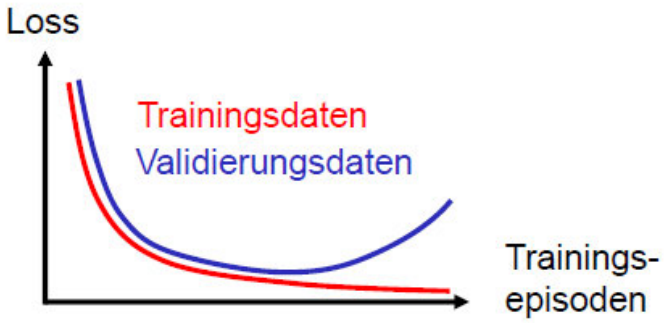


2-17 Vergleich verschiedener Optimizer *(Kingma, et al., 2017)*

Overfitting

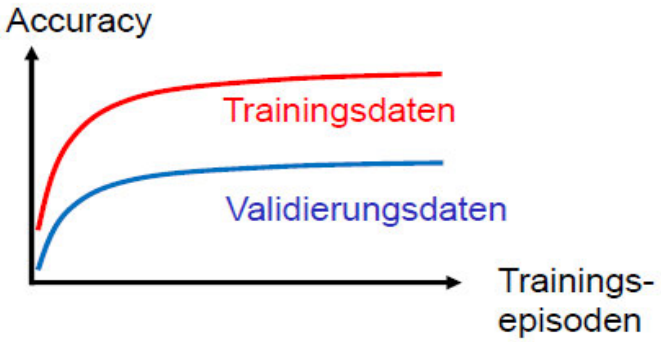
Overfitting (deutsch: Überanpassung) beschreibt eine zu starke Spezialisierung an die Trainingsdaten. Ein Datensatz sollte immer aufgeteilt werden, in einen Datensatz für das Training und einen für die Validierung. Dadurch wird die Generalisierungsfähigkeit des Neuronalen Netzes geprüft. Spezialisiert sich ein Netz zu sehr an die Trainingsdaten, verschlechtert sich die Generalisierung und das Netz kann nicht für beliebige Daten verwendet werden.

Overfitting lässt sich oftmals an den Loss- und Accuracy Metriken erkennen. Die Loss-Metrik zeigt den Loss-Wert nach jeder Epoche für den Trainings- und Validierungsdatensatz an. Die Accuracy-Metrik, oder Genauigkeit, zeigt den Anteil an korrekten Vorhersagen, zu allen Vorhersagen, die das Netz getroffen hat. Dabei ist das Ziel des Loss, so gering wie möglich und der Accuracy, so hoch wie möglich zu werden. Wenn, anhand dieser Metriken, festgestellt wird, dass der Loss der Trainingsdaten geringer wird, während der Loss der Validierungsdaten steigt, deutet das auf Overfitting an die Trainingsdaten hin. Ein graphisches Beispiel ist in Abbildung *2-18* dargestellt.



2-18 Overfitting anhand der Loss Metriken *(Meisel, 2021)*

Bei der Accuracy ist es andersherum. Ist die Accuracy der Trainingsdaten deutlich höher als die der Validierungsdaten, ist dies ein Indiz für Overfitting. Dies ist in Abbildung *2-19* an einem graphischen Beispiel zu sehen.



2-19 Overfitting anhand der Accuracy Metriken *(Meisel, 2021)*

Netz verkleinern

Ein kleineres Netz, bedeutet eine kleinere Anzahl an Neuronen. Das Training wird schwächer und die Spezialisierung auf die Trainingsdaten ebenso.

Early stopping

Umso länger ein Netz trainiert, desto stärker spezialisiert sich dieses an die Trainingsdaten. Um die zu verhindern, wird ein Training früher beendet. Ein Richtwert dafür kann aus der graphischen Darstellung der Loss-Metriken entnommen werden. An dem Punkt, wo Trainings- und Validierungsloss voneinander divergieren, sollte das Training gestoppt werden.

Regularisierung

Die Fehlerfunktion wird mit einem Regularisierungsterm erweitert, welcher hohe Gewichtswerte bestraft. Dabei sorgt dieser, bei der Aktualisierung der Gewichte, dafür, dass das vorherige Gewicht nicht zu groß werden kann.

Dropout Layer

Ein Dropout Layer führt dazu, dass ein zufälliger Teil der Neuronen, in einem Trainingsschritt, nicht trainiert wird. Dies ist zu vergleichen mit einer Verkleinerung des Netzes. Dabei kann der Anteil der Neuronen, die nicht trainiert werden sollen, als Prozentsatz gesetzt werden.

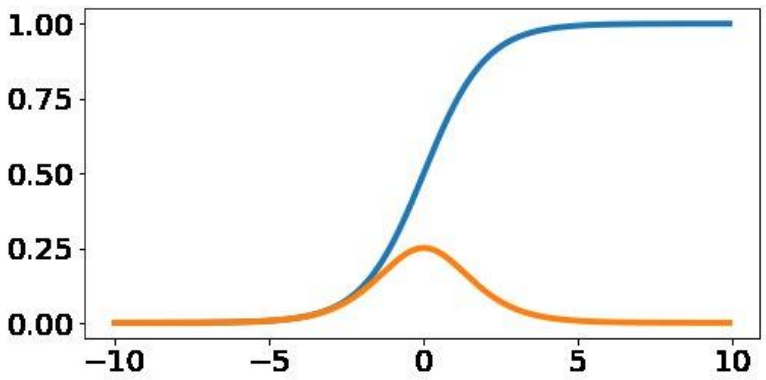
Trainingsdatensatz vergrößern

Mehr Daten zu beschaffen ist in der Regel keine leichte Aufgabe, weshalb dieser Lösungsansatz am schwierigsten umzusetzen ist. Mehr Daten zum Trainieren zu haben, die optimalerweise auch eine hohe Vielfalt aufweisen, würde das Netz an mehr Beispielen und möglichen Daten trainieren lassen, was die Generalisierung steigert.

Vanishing gradient

Bei steigender Größe und somit Tiefe eines Netzes kann es dazu kommen, dass der Gradient immer kleiner wird, wodurch das Training stark verlangsamt wird oder gar nicht vorankommt. Am Beispiel der logistischen Funktion sigmoid ist dies zu erkennen.

Wie in Abschnitt *2.3.4* erklärt, wird die Ableitung der Aktvierungsfunktion bei der Backpropagation genutzt. Betrachtet man die Ableitung der sigmoid-Funktion, in Abbildung *2-20*, ist zu sehen, dass ihr Maximum bei 0,25 liegt.

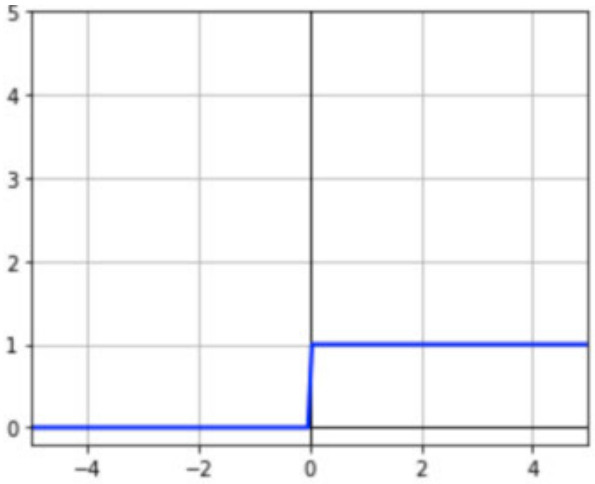


2-20 Sigmoid und dessen Ableitung in graphischer Darstellung *(Linux User, 2018)*

Diese Ableitung wird mit den bereits kleinen Gewichtswerten für jede Schicht des Netzes multipliziert , wodurch der Gradient mit der Zunahme an Schichten immer kleiner wird.

Rectified linear unit (ReLU)

Die bereits in Abschnitt *2.3.2* vorgestellte ReLU-Aktivierungsfunktion hilft dabei, das Problem zu lösen, da ihre Ableitung, zu sehen in Abbildung *2-21*, ein Maximum von 1 hat. Dadurch wird der Gradient nicht mit jeder Schicht verkleinert.



2-21 Ableitung der ReLU Funktion in graphischer Darstellung *(Anonym, Zugriff: 2022)*

BatchNormalization

Ein Problem bei der ReLU Funktion ist allerdings, dass bei der nicht-abgeleiteten ReLU Funktion eine ungleiche Verteilung der Werte entsteht. Grund dafür ist, dass der Wertebereich von 0 bis zum Wert des Eingangs gehen kann. Dabei geht die Standardisierung der Daten verloren, also der Mittelwert ist nicht mehr 0 und die Standardabweichung nicht mehr 1.

Dazu kann die BatchNormalization verwendet werden, die zwischen jeder Schicht, für jedes Neuron und pro Minibatch die Daten standardisiert. Die BatchNormalization hat eine Selbstoptimierende Eigenschaft, wodurch die optimalen Werte im Training eingestellt werden.

Ein **Minibatch** beschreibt eine zufällige Teilmenge des Datensatzes, die durch das Netz geht, bevor die Gewichte aktualisiert werden. Dabei wird der gesamte Datensatz in diesen Teilmengen durchlaufen, für jede Teilmenge werden die Fehlergradienten gemerkt und am Ende des Datensatzes werden diese genutzt, um die Gewichte zu aktualisieren.

Scaled exponential lienar unit (SeLU)

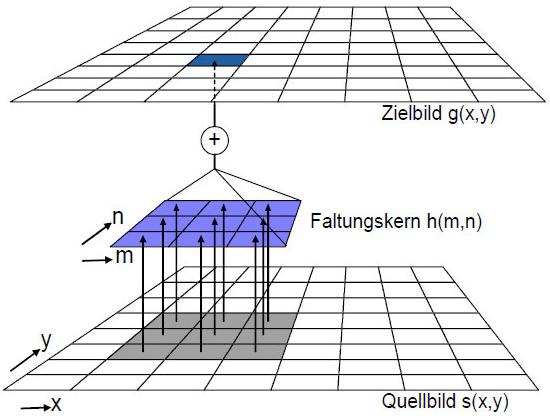
Ander als ReLU, sind hier alle Werte für $z < 0$ nicht gleich 0, sondern gleich $\alpha(e^z - 1)$. Zusätzlich werden die Werte für $z < 0$ und $z \geq 0$ mit einem Parameter λ multipliziert. Mit diesen Veränderungen werden standardisierte Eingaben, wieder als standardisiert ausgegeben.

2.3.6 Convolutional Neural Networks

Convolutional Neural Networks (CNN) (deutsch: Faltungsnetzwerke) sind in der Bildverarbeitung von großer Bedeutung und der Standard im Training von Netzwerken, die mit Bilddaten arbeiten.

Faltung

Der Grundgedanke bei der Faltung, ist die Berücksichtigung der nebenliegenden Bildpunkte, eines zweidimensionalen Bildes. Dazu wird ein Faltungskern genutzt, welcher aus einer Matrix besteht und in der Regel ungerade Dimensionen hat. Dieser wird dann über jeden Bildpunkt geführt, um die Faltung auszuführen. Dabei werden die Bildpunkte jeweils mit dem Wert im Faltungskern multipliziert, aufaddiert und durch die Anzahl der Bildpunkte geteilt. Oftmals wird dies auch als Filterung und der Faltungskern als Filterkern bezeichnet.



2-22 Vorgang bei einer Faltung *(Meisel, 2021)*

Größe des Faltungskerns

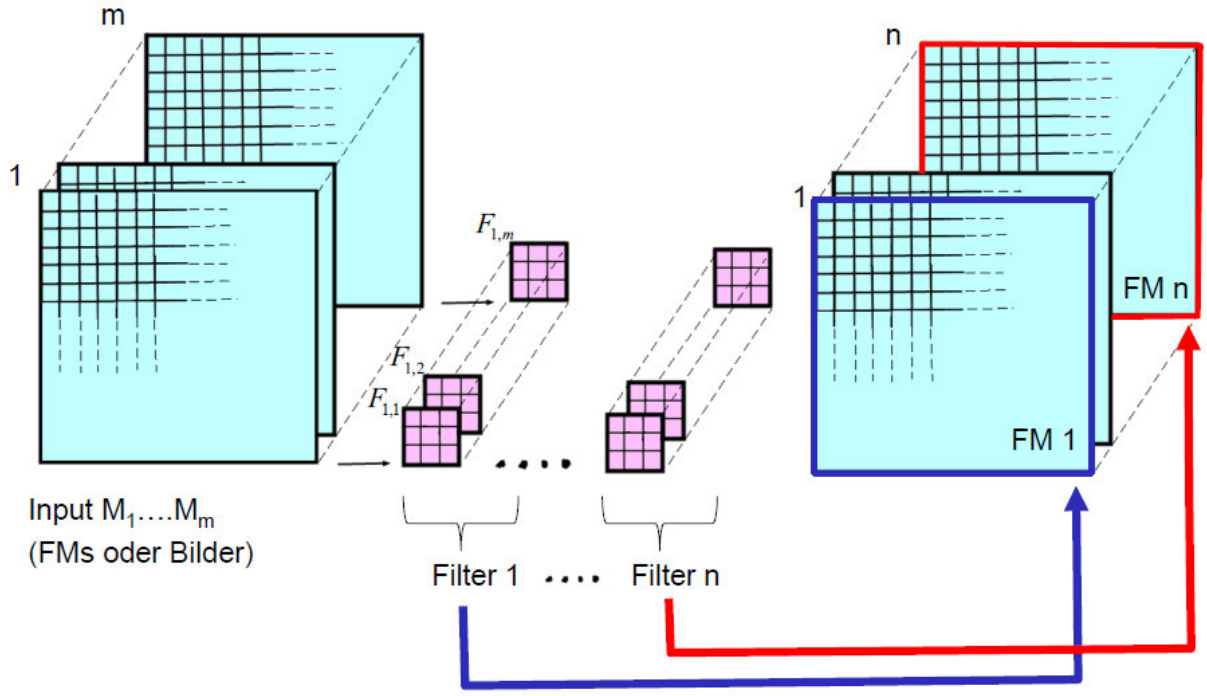
Die Größe des Faltungskerns wird üblicherweise klein gehalten, da dies Rechenaufwand, Dauer des Trainings und Speichernutzung der GPU reduziert. Ein Faltungskern nutzt einen Satz Pixel bei der Faltung, dessen Gewichte miteinander geteilt werden, sodass insgesamt weniger Gewichte bei einem kleineren Kern genutzt werden. Zusätzlich betrachtet ein kleinerer Kern nur lokale Bereiche in einem Bild und kann dadurch komplexere Merkmale erkennen und extrahieren, während ein größerer Filter gröbere und allgemeinere Merkmale erkennt und extrahiert. Dafür kann dieser möglicherweise, pro Faltung, mehr Informationen gewinnen.

Ungerade Filtergrößen wie 3x3 oder 5x5, im Gegensatz zu 2x2 oder 4x4, erlauben alle Pixel, um den Ausgabe Pixel herum, symmetrisch verarbeitet zu werden. Aus diesen Gründen hat sich 3x3 als Standardgröße bewährt.

Die Größe 1x1 wird zur Reduktion der Dimensionen genutzt. Ein 1x1 Kernel kann dann die Tiefe beziehungsweise Menge von Feature-Maps (Vorgriff auf *2.3.6*) auf eins reduzieren, ohne die Größe des Bildes zu verändern. Daher kann ein 1x1 Kernel im Code-Layer eines Autoencoders (Vorgriff auf *2.3.7*) genutzt werden.

Feature-Maps

Als Feature-Maps werden die Ausgabebilder nach der Faltung bezeichnet. Der Faltungskern entspricht hier einem Neuron und wird durch das Training erzeugt und verändert. Bei einem CNN können mehrere Bilder eingespeist werden, die mit jeweils einem Faltungskern gefaltet werden. Dabei teilen sich die einzelnen Faltungskerne die Gewichte, sodass ein Neuron mehreren Faltungskernen entspricht und pro Faltungskern eine Feature-Map entsteht. Die Anzahl an Neuronen und somit unterschiedlicher Faltungskerne oder Filter bestimmt daher die Anzahl an Ausgabe-Feature-Maps. Ein Layer, welcher die Faltung ausführt, wird als Convolutional-Layer bezeichnet.



2-23 Anwendung der Faltungskerne auf die Eingabebilder *(Meisel, 2021)*

Stride

Stride steht für die Schrittgröße des Faltungskerns bei der Anwendung auf ein Bild. Bei einem Stride von 1 bewegt sich der Faltungskern nur um einen Schritt von links nach rechts, bis das Ende des Bildes erreicht wurde. Anschließend bewegt er sich, vom Anfang aus, einen Schritt nach unten und wieder nach rechts. Da der Faltungskern die umliegenden Bildpunkte nutzt, um einen festzulegen, fällt der Rand eines Bildes weg und die Feature-Map ist kleiner.

Je größer daher der Stride ist, desto kleiner wird die Feature-Map bei der Ausgabe. Ein Stride von 1 macht beispielsweise aus einem 7x7 Bild, eine 5x5 Feature-Map, während ein Stride von 2 eine 3x3 Feature-Map ausgibt.

Zeropadding

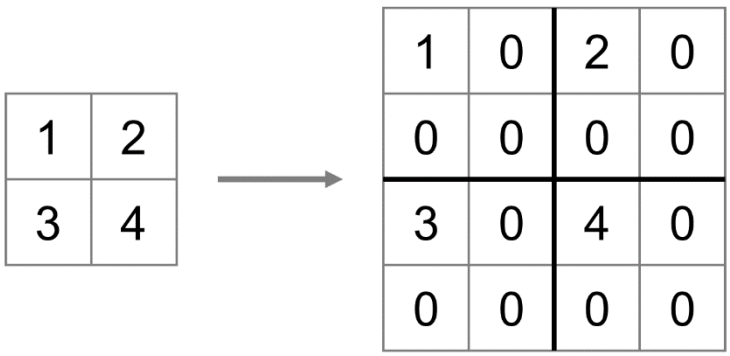
Um diese Verkleinerung zu umgehen, kann ein Rand eingefügt werden, der mit Nullen gefüllt ist und das Bild vergrößert. Die Nullen führen bei der Faltung zu keinem Unterschied, da diese mitaddiert werden. Der Rand wird dann bei einem Stride von 1 entfernt und die Größe der Feature-Map entspricht der, des Eingabebildes.

Pooling

Eine andere Art, um die Bildgröße zu verkleinern, ist das Pooling. Hierbei wird ebenso ein Faltungskern genutzt, oftmals mit einer Größe von 2x2 und einem Stride von 2, wodurch die Feature-Map halb so groß wird. Dabei gibt es verschiedene Möglichkeiten, welcher Wert aus der Faltung in das Zielbild übernommen wird. Die gängigste Variante ist das Max-Pooling. Dabei wird der Bildpunkt mit dem größten Wert in das Zielbild übernommen. Das Pooling wird als Layer realisiert.

Upsampling

Zum Vergrößern der Feature-Maps gibt es zwei Varianten, der Upsampling-Layer oder der Transposed Convolutional-Layer. Der Upsampling-Layer funktioniert sehr ähnlich wie der Pooling-Layer. Soll ein Bild beispielsweise wieder in der Größe verdoppelt werden, wird für jeden Bildpunkt eine 2x2 Repräsentation dessen gespeichert. Dabei entspricht einer der Werte, der des Quellbildpunktes und die restlichen drei sind entweder 0 oder 1.
Der Transposed Convolutional-Layer nutzt einen Stride, um das Bild zu vergrößern. Dabei bedeutet Stride, anders als im Convolutional-Layer, nicht die Schrittweite, sondern das Zeropadding zwischen den Bildpunkten.



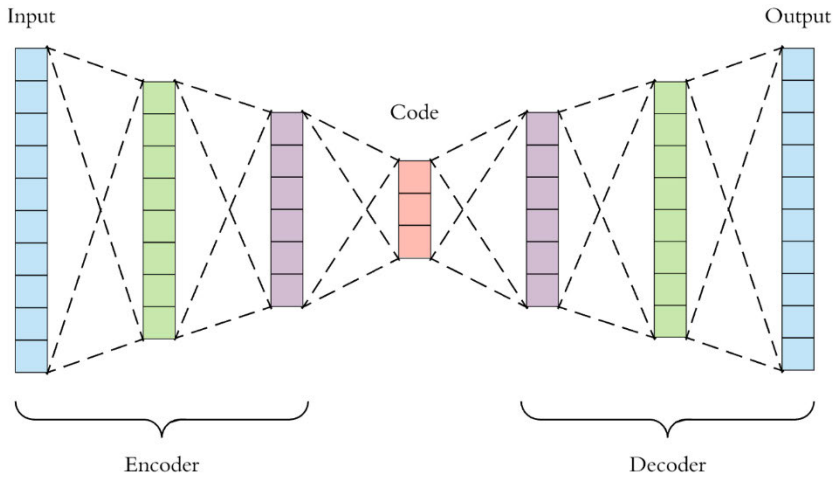
2-24 Beispiel zu 2D-Upsampling *(Djib2011, 2018)*

Merkmale

Durch die mehreren Schichten eines CNN werden auf verschiedenen Ebenen, verschiedene Merkmale gelernt. Auf den ersten Schichten werden einfache Merkmale erlernt, wie Punkte oder Kanten. Die darauffolgenden Schichten erlernen immer komplexere Merkmale, aufbauend auf denen, die von den vorherigen Schichten erlernt wurden. Das ermöglicht dem CNN, Details und komplexe Merkmale aus Bildern zu entnehmen und macht Aufgaben, wie Objekterkennung oder Bildklassifizierung, möglich.

2.3.7 Autoencoder

Autoencoder sind eine Art Neuronales Netz. Anders als bei den bisher vorgestellten Netzen ist hier der Ausgang gleich dem Eingang. Bei CNNs, die beispielsweise zur Klassifizierung genutzt werden, werden die Dimensionen eines Bildes durch das Netz verkleinert und in der letzten Schicht komplett flach ausgegeben. Ein Autoencoder wiederum verkleinert die Dimensionen, in einem Encoder Teil, erlernt die niedrigdimensionale Repräsentation in einem Code-Layer, oder auch Bottleneck, und vergrößert die Dimensionen zu ihrem Ursprung in einem Decoder Teil.



2-25 Aufbau eines Autoencoders *(Dertat, 2017)*

Der Vorteil hierbei ist, dass keine Labeldaten gebraucht werden, da die Eingabedaten gleich Labeldaten sind. Dadurch kann auch die Aufgabe einer Klassifizierung aufgeteilt werden. Zuerst wird ein Autoencoder genutzt, welcher mit vielen Daten ohne Label lernt und anschließend ein Klassifizierer, welcher mit weniger Daten mit Label lernt.

Bei Autoencodern kommt das Overfitting Problem wieder besonders auf, weil die Gefahr besteht, dass der Autoencoder einfach den Eingang auf den Ausgang kopiert. Dafür gibt es verschiedene Arten von Autoencodern, die versuchen den Autoencoder komplex genug zu gestalten, sodass dieser aus der niedrigdimensionalen Repräsentation der Daten lernt, aber auch nicht zu komplex ist, um Overfitting zu vermeiden.

Undercomplete Autoencoder

Der Bottleneck hat bei dieser Art von Autoencoder eine viel kleinere Dimensionsgröße als die Eingabe. Die Komplexitätsstufe dieses Autoencoders ist dadurch sehr gering und die Gefahr von Overfitting ist ebenfalls hoch.

Denoising Autoencoder

Die Eingangsdaten werden verrauscht, sodass der Bottleneck nicht mehr so klein sein muss. Einfaches Kopieren der Eingangsdaten auf die Ausgangsdaten ist nicht mehr möglich und das Netz muss nach besonderen Eigenschaften suchen.

Sparse Autoencoder

Der Bottleneck wird hierbei nicht durch eine geringere Dimension oder Neuronen realisiert, sondern durch eine Regularisierung. Anders als die Regularisierung aus 2.3.5, bestraft diese hohe Ausgabewerte, statt hohe Gewichtswerte. Dabei findet eine Selbstoptimierung anhand der Daten statt und komplexere Aufgaben sind möglich.

CNN Autoencoder

Dieser Autoencoder unterscheidet sich nur in der Dimensionalität der Eingangsdaten vom Rest. Wie bei einem CNN, akzeptiert dieser zweidimensionale Bilder und führt die Faltung auf diese aus. Durch Pooling oder Stride kann hier die Dimension verkleinert und mit Upsampling oder dem Transposed Convolutional Layer wieder vergrößert werden.

Variational Autoencoder

Bei diesem Autoencoder wird eine besondere Loss Funktion verwendet, welche eine Standardisierung der Daten im Bottleneck nutzt. Dies führt zu einer zentrierten Streuung der Daten, sodass eine gute Rekonstruktion der Daten im Decoder stattfinden kann.

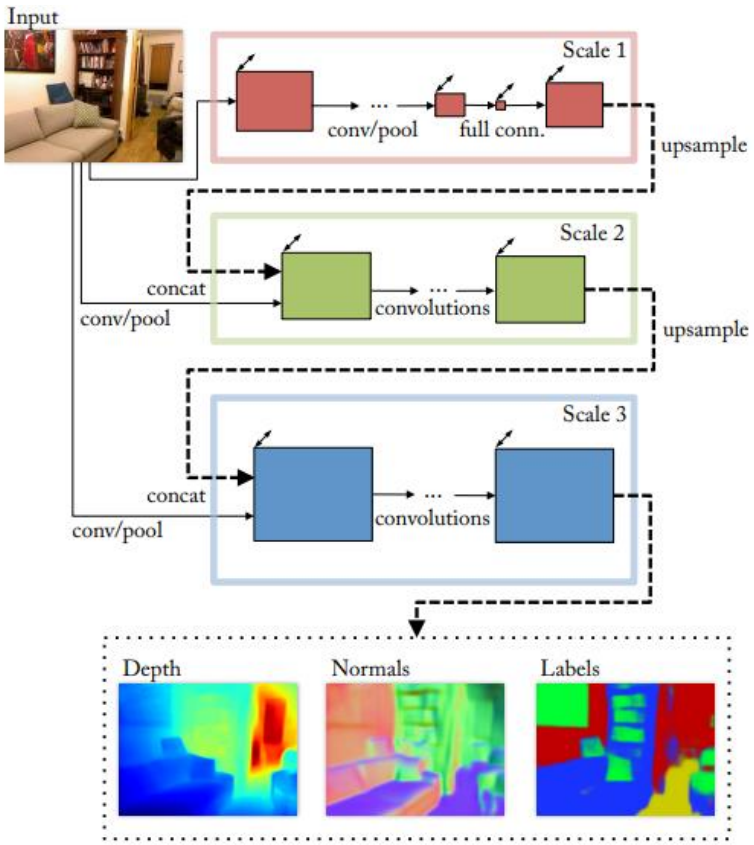
Das Ziel dieser Arbeit ist es, Tiefenbilder aus RGB-Bildern zu rekonstruieren. Dabei werden die Dimensionen eines RGB-Bildes reduziert und anschließend wieder erhöht, um ein Tiefenbild zu erstellen. Daher wird eine Encoder-Decoder Struktur benötigt, weshalb für diese Arbeit ein Autoencoder entwickelt wurde.

3 Stand der Technik

In diesem Kapitel werden zusammengefasst Techniken und Veröffentlichungen vorgestellt, die eine Relevanz zu dieser Arbeit haben. Verweise auf die Quellen der Arbeiten werden angegeben und können genutzt werden, um tiefer in die Veröffentlichung einzusteigen.

3.1 Tiefenvorhersage anhand einer „Multi-scale“ Architektur

Mit 2352 Zitierungen (Google, Zugriff: 2022), ist diese eine der womöglich meistzitierten Veröffentlichungen zum Thema Tiefenbildrekonstruktion, oder auch Tiefenvorhersage. Thema dieser Arbeit war, die Entwicklung eines Modells, welches aus einem RBG-Bild, ein Tiefenbild, ein Normalen Bild und ein Bild zur semantischen Gruppierung verschiedener Objekte erstellt. Dazu wurde eine „Multi-Scale Convolutional Architecture“ entwickelt, welche in Abbildung *3-1* zu sehen ist.



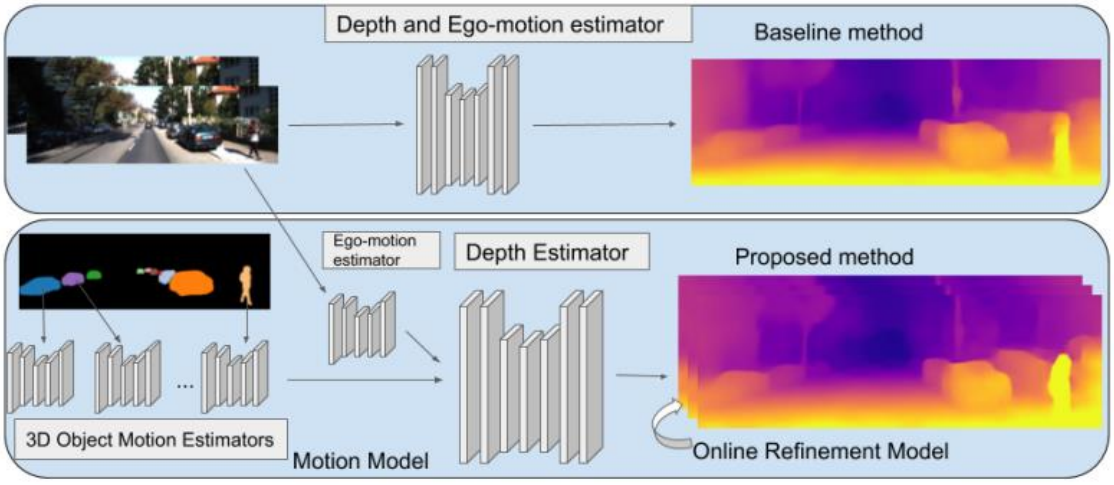
3-1 Multi-Scale Architektur aus *(Eigen, et al., 2015)*

Die Architektur nutzt drei verschiedene Skalen, um verschiedene Merkmale aus den Bildern zu entnehmen:
Auf der ersten Skala wird das komplette Bild genutzt, um grobe Merkmale zu entnehmen. In Skala zwei wird eine schmalere, aber dafür detailliertere Sicht des Bildes genutzt, um auf einem mittelgroßen Bild Merkmale zu extrahieren. Zusätzlich werden die Informationen aus der ersten Skala, durch Konkatenation, inkorporiert. Die dritte Skala vergrößert die Auflösung dieser Merkmale wieder und bindet, auch hier, die Informationen der vorherigen Skala mit ein. Diese Architektur wird jeweils genutzt, um die Tiefe, die Normalen und semantische Gruppierung zu erstellen, indem für jedes der drei Aufgaben eine separate Loss Funktion verwendet wurde.

Weitere Informationen und Vorstellung der Ergebnisse sind zu finden unter (Eigen, et al., 2015).

3.2 Erstellen von Tiefenbildern ohne Sensoren

In dieser Veröffentlichung wird das Problem der Datenbeschaffung, in Bezug auf Tiefenbilder behandelt. Um die Tiefe zu messen, werden oftmals teure Sensoren genutzt, die nicht für jeden erwerbbar sind, möglicherweise fehlerhafte Messungen produzieren und Arbeitskräfte zur Bedienung benötigen. Daher wird in dieser Arbeit eine Variante der Datenbeschaffung, mittels Modellierung der dreidimensionalen Bewegungen verschiedener Objekte und der Kamera, anhand von monokularer Videoaufnahmen, vorgestellt. Die Aufgabe kann in zwei Teile aufgeteilt werden, Einschätzung der Tiefe und der 3D-Bewegung der Kamera.
Die Tiefe wird mit einer Autoencoder Architektur realisiert, welche aus einem RGB-Bild die Tiefe ermittelt. Für die Bewegung der Kamera wird eine Sequenz an Bildern genutzt, aus welchen die Unterschiede genutzt werden, um einen sechsdimensionalen Vektor zu erlangen, welcher die Position und Rotation im 3D repräsentiert.



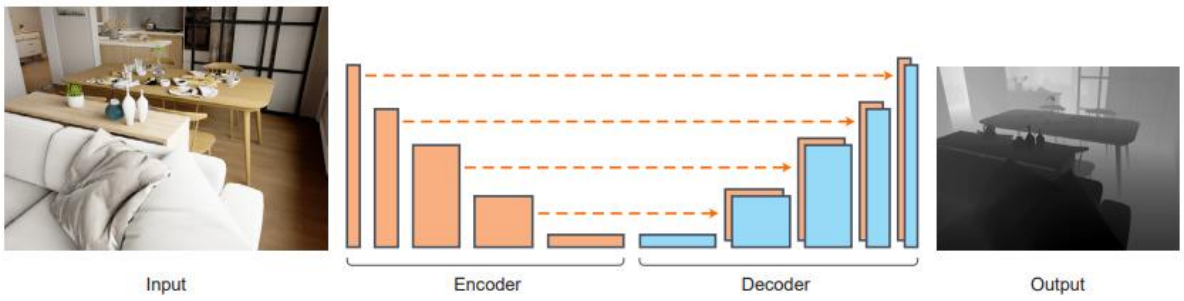
3-2 Depth and Ego-motion estimator und Motion Model aus *(Casser, et al., 2018)*

In Abbildung *3-2* ist die Repräsentation der beiden Aufgaben als der „Depth and Ego-motion estimator“ dargestellt. In dieser Arbeit wurde darauf aufgebaut und die Bewegungen einzelner Objekte im 3D miteinbezogen. Zusätzlich zu der RGB-Bildsequenz ist nun eine Instanz-segmentierung in Form einer Maske dazugekommen, in der die vorhandenen Objekte maskiert werden. Diese werden später hinzugefügt, um zu bestimmen, wie sich die Kamera bewegt haben muss, sodass die Objekte nun woanders auftauchen. Im „Online Refinement Model“ werden während des Trainings, Optimierungen auf den Daten ausgeführt und dem Modell neu eingespeist. Zusammen ergeben die einzelnen Komponenten das in dieser Arbeit entwickelte, „Motion Model“ zum Erstellen von Tiefenbildern.

Mehr Details sowie die Vorstellung der Ergebnisse können in (Casser, et al., 2018) nachgelesen werden.

3.3 Hochqualitative Tiefeneinschätzung anhand von Transfer Learning

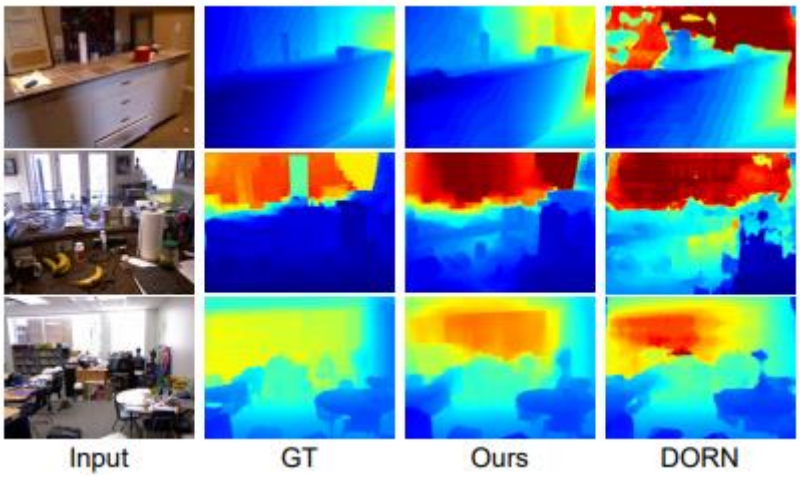
Das in dieser Arbeit veröffentlichte Modell nutzt ein vortrainiertes Modell in einer Autoencoder Architektur, um Tiefenbilder zu erstellen. Dabei wurde als Ziel gesetzt, die Architektur relativ einfach zu halten und weiterhin hochqualitative Ergebnisse zu erzielen. Hierbei wurde der Ansatz verfolgt, für den Encoder Teil ein vortrainiertes Netz zu nutzen. Zusätzlich wurden Skip Connections eingebaut zwischen Encoder und Decoder, um erlernte Informationen beizu-behalten. Die Architektur ist in Abbildung *3-3* zu sehen.



3-3 Modellarchitektur aus *(Alhashim, et al., 2019)*

Das vortrainierte DenseNet-169 wurde gekürzt und ansonsten unmodifiziert als Encoder geneutzt. Der Decoder besteht aus simplen Convolutional-Blöcken, bestehend aus jeweils zwei Convolutional-Layer und einem Layer zur Konkatenation. Der Decoder wird nicht gespiegelt zum Encoder dargestellt und enthält deutlich weniger Layer als das DenseNet-169.
Die genutzte Loss Funktion besteht aus einer Kombination von drei, jeweils gewichteten, Loss Funktionen L_{depth}, L_{grad} und L_{SSIM}. Dieser relativ simple Ansatz hat eine Verbesserung zu den

Ergebnissen des neuesten Standes gezeigt, was in Abbildung *3-4* zu sehen ist. Zu vermerken ist, dass der neueste Stand sich auf 2019, das Jahr der Veröffentlichung, bezieht.



3-4 Ergebnisse im Vergleich zum Stand der Technik aus *(Alhashim, et al., 2019)*

Mehr zur Implementation und zu den Ergebnissen ist bei (Alhashim, et al., 2019) zu finden.

4 Arbeitsumgebung und verwendete Ressourcen

Die gesamte Implementierung und Untersuchung wurde auf dem, von Project Jupyter (Project Jupyter., Zugriff: 2020) entwickelten, JupyterHub (HAW Hamburg., Zugriff: 2020) durchgeführt, der den Studierenden und Mitarbeitern der Hochschule für Angewandte Wissenschaften Hamburg zur Verfügung gestellt wird. Auf JupyterHub können Webanwendungen, sogenannte Jupyter Notebooks, verwendet werden. Ein Jupyter Notebook kann mit verschiedenen Kernels genutzt werden, um verschiedene Programmiersprachen zu unterstützen. Der übliche Gebrauch ist hierbei die Programmierung von Python Projekten, da der Kernel für Python direkt mit ausgeliefert wird (erentar., 2022).

4.1 Hardware

Grafikkarte: Nvidia Tesla V100
Grafikkartenspeicher: 16GB HBM2

Prozessor: Intel® Xeon® Gold
Geschwindigkeit: 2.4 GHz

Festplattenspeicher: 700GB. Erweitert nach Anfrage.

4.2 Software

Python

Zur Entwicklung wurde die Programmiersprache Python 3.9.7 (Python., 2021) genutzt. Durch dessen breite Auswahl an Frameworks und Bibliotheken zum Machine Learning hat sich Python als der Standard im Bereich der KI und Machine Learning etabliert. Einfache Lesbarkeit und Syntax, dynamische Typisierung sowie schnelle Entwicklung machen Python zu einer mächtigen Programmiersprache für neue und erfahrene Entwickler.

Keras

Keras (Keras., 2015), veröffentlicht von Francois Chollet, ist eine Deep-Learning Bibliothek, die für jeden frei zugänglich ist. Keras wurde in Python geschrieben und wird in dieser Arbeit für die Entwicklung der Modelle genutzt.

Keras bietet seine high-level API an und fokussiert sich dabei auf Modularität, Benutzerfreundlichkeit und Erweiterbarkeit. Aufbauend auf TensorFlow 2 ist Keras, mit mehr als einer Millionen Nutzern [Stand: 2021 (Keras, Zugriff: 2022)], eines der beliebtesten Tools im Bereich Machine Learning. Eine große Besonderheit von Keras ist das Spektrum der Komplexität bei der Implementierung von Modellen. Von simplen, sogenannten Sequential Modellen, die in wenigen Zeilen Code erstellt werden können, bis zu benutzerdefinierten Modellen, aus modifizierbaren Klassen, bietet Keras die Möglichkeit, Modelle jeglicher Art zu erstellen.
Die komplette Entwicklung und das Training des Modelles dieser Arbeit wurde mit Hilfe der Keras Functional API, welche flexibleres gestalten von Modellen erlaubt, durchgeführt.

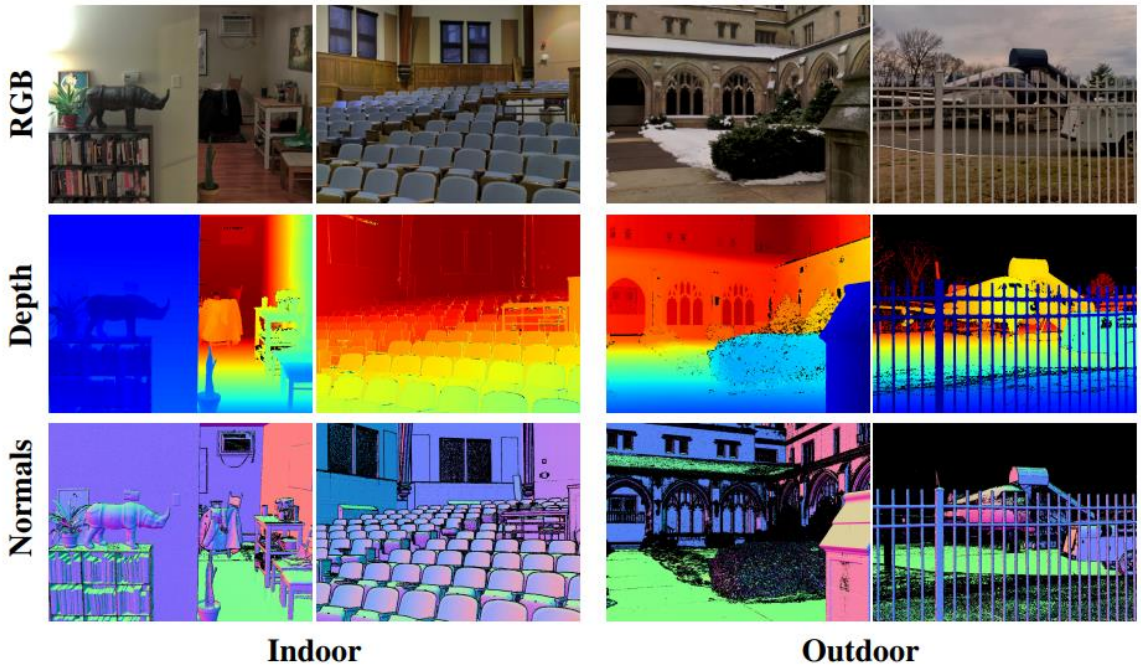
Weitere verwendete Python Bibliotheken

- *Numpy*: Speichern und Verarbeiten von Bilddaten in mehrdimensionalen Arrays
- *Matplotlib*: Visualisierung von Bilddaten als Graphen oder RGB-Bild
- *OpenCV*: Lesen und Bearbeiten von Bilddateien

4.3 Datensatz

Für die monokulare Tiefenbildrekonstruktion gibt es noch nicht viele Datensätze, wie im Vergleich zur Bildklassifizierung oder Objekterkennung (Papers with Code, Zugriff: 2022). Die mit Abstand beliebtesten Datensätze, KITTI (Geiger, et al., Zugriff: 2022) und NYU-Depth V2 (Silberman, et al., 2012), werden in zahlreichen Veröffentlichungen genutzt und haben einen großen Datenbestand. Der KITTI Datensatz beinhaltet über 94.000 Bilder mit entsprechenden Tiefenbildern. Diese wurden mit einem Velodyne VLP-64 LiDAR Sensor aufgenommen und beinhalten Bilder von verschiedenen Straßen.
Der NYU-Depth V2 Datensatz beinhaltet mehr als 400.000 Bilder und Tiefenbilder. Zur Aufnahme der Bilder sowie Messung der Tiefe von verschiedenen Innenräumen, wurde die von Microsoft entwickelte Kinect (Microsoft Xbox, Zugriff: 2020) Kamera genutzt.

Der, für diese Arbeit benutzte, Datensatz DIODE (Vasilijevic, et al., 2019) besteht aus Bildern von Innen- und Außenräumen. Mit etwas über 26.000 Bildern und Tiefenbildern ist dieser Datensatz deutlich kleiner als die Konkurrenten, zeichnet sich jedoch darin aus, dass dieser eine höhere Vielfalt und Auflösung hat. Für die Tiefenmessungen wurde ein FARO Focus S350 LiDAR Sensor genutzt, welcher einen phase-shift Laserscanner nutzt. Dabei wird eine Lichtwelle, oder auch Laserwelle, auf ein Objekt gesendet, welches diese reflektiert. Die Reflektion wird vom Sensor empfangen und die Verschiebung der Phase, also der Unterschied zwischen der gesendeten und der empfangenen Lichtwelle, schließt auf die Entfernung hin. Das erlaubt dem Sensor hochauflösende Messungen von Innen- und Außenräumen und hat somit einen Vorteil gegenüber den Messverfahren von KITTI und NYU Depth V2. Des Weiteren war es, dank der hohen Qualität der Aufnahmen, möglich, die Normalen der Ebenen zu ermitteln und dem Datensatz beizufügen.



4-1 Beispiele aus dem DIODE Datensatz für Tiefe und Normalen *(Vasilijevic, et al., 2019)*

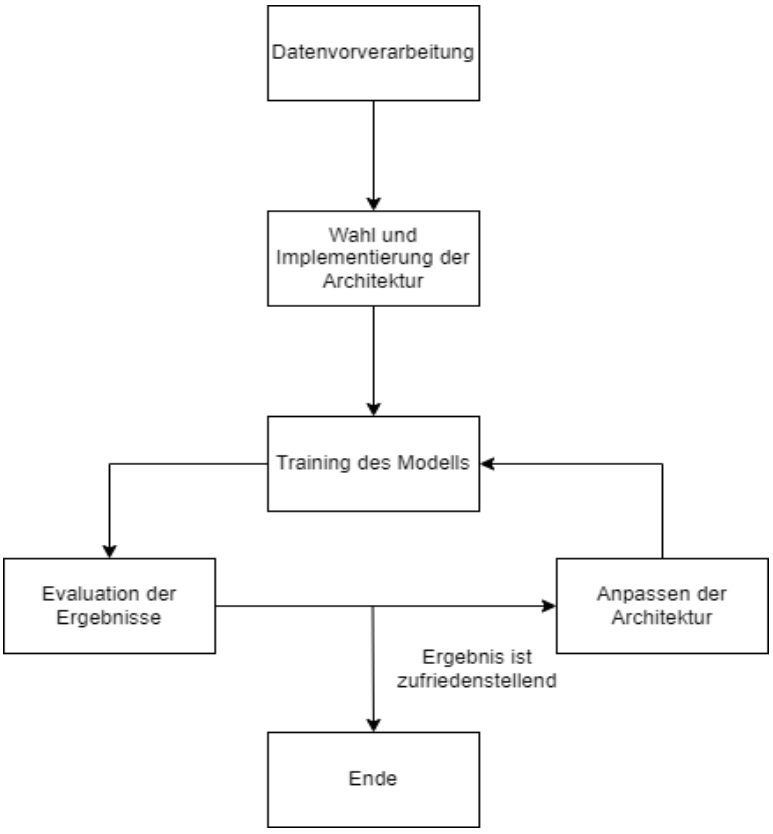
Zusätzlich zu den zuvor genannten Vorteilen ist der DIODE Datensatz noch relativ ununtersucht, weshalb dieser für diese Arbeit verwendet wurde. Hierbei wurde der Fokus allerdings auf die Innenräume gelegt. Einige der zuvor erwähnten Details zum DIODE Datensatz sowie Experimente und Ergebnisse zu dem Datensatz sind in (Vasilijevic, et al., 2019) zu finden.

5 Implementierung und Evaluation

5.1 Einstieg in Autoencoder Bildrekonstruktion

Der erste Schritt der Arbeit, nach inhaltlicher Einarbeitung, war der Einstieg in die Implementierung von Autoencodern. Um die grundlegende Funktionsweise sowie das Verhalten bei verschiedenen Parametern zu untersuchen, wurde als Ziel des Modells die reine Bildrekonstruktion gesetzt. Somit entsprachen die Eingabe- und Labeldaten, den RGB-Bildern des Datensatzes. Dabei wurden verschiedene Ansätze getestet, um das beste Ergebnis zu erreichen. Die theoretische Herangehensweise und die Erkenntnisse, zusammen mit der Implementierung und den Ergebnissen, werden in diesem Kapitel vorgestellt.

5.1.1 Arbeitsablauf



5-1 Arbeitsablauf Bildrekonstruktion

Im ersten Schritt werden die Daten vorverarbeitet. Die Eingabebilder des Modells und die Labeldaten sind gleich, weshalb die Vorverarbeitung nur einmal für beide durchgeführt wird. Die Vorverarbeitung bleibt relativ einfach und beinhaltet nur die Skalierung, Normalisierung und Konvertierung der Bilder.

Im zweiten Schritt wird die Architektur des Modells gewählt, das Modell trainiert und das Ergebnis im nächsten Schritt evaluiert. Die Evaluation bezieht sich zum einen auf die Loss-Metriken des Trainings- und Validierungsdatensatzes und zum anderen auf die eigene subjektive Wahrnehmung der Ergebnisse. Im darauffolgenden Schritt wird das Modell anhand dieser Evaluation angepasst. Dies umfasst Anpassung jeglicher Hyperparameter, wie die Art und Anzahl der Layer, der Loss Funktion, des Optimierers, der Epochen, Batch Size, Größe und Art des Bottlenecks. Zyklisch werden die Ergebnisse dieser Veränderungen evaluiert und weiterhin Anpassungen durchgeführt, bis ein zufriedenstellendes Ergebnis erzielt wird und genug Erkenntnisse gewonnen werden. Aus dem Zyklus werden verschiedene Erkenntnisse aus der Implementierung und Untersuchung des Modells, als Szenarien, entnommen, miteinander verglichen und bewertet, was in Abschnitt *5.1.4* zu finden ist.

5.1.2 Datenvorverarbeitung

Dieser Abschnitt beschreibt die Schritte der Datenvorverarbeitung und zeigt einige Beispiele der Arten der Datenvorverarbeitung und ihren Nutzen. Zuerst werden die Bilder in das Programm geladen, dann wird das Format des Bildes angepasst. Anschließend wird das Bild normalisiert, verkleinert und in einen TensorFlow Datentyp konvertiert.

Arten von Datenvorverarbeitung

Bei der Datenvorverarbeitung gibt es etliche Vorgehensweisen, wie das Bild neu zu skalieren, zu normalisieren oder die Augmentation von Daten. Dies sind nur einige Beispiele, da die Wahl der Vorgehensweise stark von den Daten und das Ziel des Trainings abhängig ist und jegliche Form annehmen kann. Zum Beispiel hilft die Augmentation von Daten dabei den Datenbestand zu erhöhen, ohne neue Daten beschaffen zu müssen. Ein Bild kann genutzt werden, welches rotiert und verschoben wird, um aus einem Datenpunkt drei zu erstellen. Datenbeschaffung stellt nämlich eine große Herausforderung beim Machine Learning dar.

Die Skalierung eines Bildes, indem dieses verkleinert wird, hilft stark dabei die Speicher-belastung zu reduzieren, da die Parameterzahl deutlich sinkt. Die Verkleinerung wird aber auch genutzt, um gewisse Merkmale der Daten nah beieinander zu haben, wodurch das Modell eine bessere Chance hat schneller und besser zu trainieren. Nimmt man das Einkaufen als Beispiel, fällt es dem Käufer deutlich leichter Gemüse miteinander zu vergleichen, wenn es direkt nebeneinander liegt. Liegt das Gemüse weit entfernt voneinander, fallen Details wie Flecken, Beulen oder Härte nicht mehr so leicht auf.

Abhängig davon, woher der Datensatz kommt, muss dieser gegebenenfalls manuell in eine Teilmenge des Trainings- und Validierungsdatensatzes aufgeteilt werden, was auch eine Art von Vorverarbeitung ist. Dabei wird oftmals ein Verhältnis von 70% für Training und 30% für Validierung empfohlen (Gholamy, et al., 2018).

Laden der Bilder

Um Bilder in Python bearbeiten und zum Trainings eines Modells nutzen zu können, müssen diese in ein Numpy-Array geladen werden. Dafür wird die Bibliothek OpenCV verwendet.

Änderung des Formats

Das Standardformat für Farbbilder ist RGB und wird somit auch bei dem Training von Modellen genutzt. Da die OpenCV Bibliothek jedoch, beim Laden der Bilder, das Format BGR verwendet, müssen diese in das RGB Format konvertiert werden.

Verkleinerung der Bilder

Die Originalgröße der Bilder des Datensatzes beträgt 1024x768 Pixel, was für ein Training sehr groß ist. Aufgrund limitiertem Grafikkartenspeicher werden die Bilder auf die Größe 128x128 Pixel verkleinert, damit das Laden von tausenden Bildern funktioniert. Zwar sind Merkmale im Bild näher beieinander, es gehen allerdings auch Details des Bildes unter, die sich potenziell als wichtig im Training herausstellen könnten. Oftmals reicht eine Größe von 128x128 jedoch aus, um zufriedenstellende Ergebnisse zu erreichen.

Normalisierung der Bilder

Wenn die Bilder in ein Numpy-Array geladen wurden, können arithmetische Operationen auf diese, und somit auf den gesamten Inhalt, ausgeführt werden. Daher können die Bilder normalisiert und deren Werte befinden sich im Bereich von 0 bis 1.

Konvertieren in einen Tensorflow Datentyp

Tensorflow hat für die primitiven Datentypen wie Integer, Floats, Booleans oder Strings einen eigenen Datentyp. Zum Training eines Modells in Keras ist es empfehlenswert, dass die Daten den Datentyp eines Tensorflow haben. Dafür kann mit der TensorFlow API das Bild konvertiert werden. Daher wird jedes Bild, als letzter Schritt, in einen 32 Bit Float von Tensorflow umgewandelt.

5.1.3 Modellarchitektur

Als Modellarchitektur wird das Konstrukt als Gesamtes bezeichnet. Das bedeutet, es beinhaltet alles von der Art und Aufbau des Modells, den Hyperparametern bis zu den Gewichten, die beim Training generiert und verändert werden. In den folgenden Abschnitten werden verschiedene Teile der Architektur eines Autoencoders erläutert und untersucht, woraus sich an eine Architektur für diesen Anwendungsfall genähert wurde.

Art des Autoencoders

Wie in Abschnitt *2.3.7* erwähnt, existieren verschiedene Arten von Autoencodern, abhängig von dem Datensatz und das Ziel des Modells. Da hier Bilddaten genutzt wurden, war die Nutzung eines Convolutional Autoencoders vorgeschrieben.

Aufbau des Autoencoders

Der Aufbau des Autoencoders beinhaltet hauptsächlich vier Aspekte: Tiefe des Autoencoders und somit Anzahl der Layer, Art und Anordnung dieser Layer, Größe der Filter und die Größe des Bottlenecks. Die Wahl aller vier dieser Aspekte spielt eine wichtige Rolle beim Entwickeln eines Autoencoders, da sie eine starke Auswirkung auf das Ergebnis haben.

Tiefe des Autoencoders

Die Tiefe des Autoencoders bestimmt die Anzahl der Neuronen und somit die Stärke des Lernens. Ein tieferes Modell lernt stärker, was aber nicht immer bedeutet, dass dies optimal ist, da es zu Overfitting oder dem vanishing gradient Problem kommen kann, die in Abschnitt *2.3.5* erläutert wurden.

Art der Layer

Während bei einem Convolutional-Autoencoder die Wahl eines Convolutional-Layers so gut wie verpflichtend ist, gibt es bei diesem Layer, in der Keras API, noch einige Parameter, die angepasst werden können. Dazu gehören die Filteranzahl, Kernelgröße, Strides, Padding, die Aktivierungsfunktion, die Initialisierung der Gewichte und einer Regularisierung. Die Funktion dieser Parameter wurde in Abschnitt *2.3.5* und *2.3.6* erklärt.

Für die **BatchNormalization** stellt die Keras API einen BatchNormalization-Layer zur Verfügung, der dessen Input normalisiert und standardisiert.

Activation-Layers können in Keras genutzt werden, um Aktivierungsfunktionen zu nutzen, die nicht als Parameter eines Convolutional Layers übergeben werden können. Diese Layer können dann auch mit ihren Parametern leicht angepasst werden.

Bei dem **Dropout-**Layer kann die Rate des Dropouts, als Parameter in einem Bereich zwischen 0 und 1 gesetzt werden.

Das **Pooling** kann mit einem Pooling-Layer realisiert werden. Parameter, wie *pooling_size*, *stride* und *padding* dieses Layers erlauben Anpassungen an der Verkleinerung. Zusätzlich gibt es noch verschiedene Arten von Pooling-Layer, die sich in den Dimensionen, die sie akzeptieren und der Art, wie das Pooling durchgeführt wird unterscheiden.

Im Gegensatz dazu gibt es den **Upsampling**-Layer, welcher anders als der Pooling Layer, nur über den "size" Parameter angepasst werden kann und die Feature-Maps wieder vergrößert. Ergänzend gibt es den Conv2DTranspose-Layer, welcher den **Transposed Convolutional-Layer** realisiert.

Concatenate-Layer werden genutzt, um die Inputs aus vorherigen Layer mit einem nachfolgenden Layer zu verbinden.

Die Keras API bietet noch viele weitere Layer. In diesem Absatz wurden die gebräuchlichen und für diese Arbeit genutzten erläutert.

Anordnung der Layer

Die Anordnung und Kombination dieser Layer müssen in gewissen Szenarien beachtet werden. Die Position der Pooling- und Upsampling-Layer muss bei der Konstruktion eines Autoencoders beachtet werden, sodass die Dimensionen der Ausgabe zu denen der Labeldaten passen. In der Regel werden die Feature-Maps, zur Verkleinerung, gespiegelt vergrößert. In dem Fall muss für jeden Pooling Layer im Encoder ein Upsampling Layer im Decoder sein. Pooling geschieht nach dem Training der Gewichte, weshalb das Upsampling vor dem Training stattfinden muss und somit vor dem Convolutional-Layer. Bei der Anordnung von Layern, wie die BatchNormalization, Aktivierungsfunktion, Dropout und Pooling gibt es keine feste Vorschrift. Ioffe und Szegedy beschreiben in (Ioffe, et al., 2015), dass der BatchNormalization Layer vor einem Layer für eine Aktivierungsfunktion positioniert sein sollte. Francois Chollet, der Entwickler von Keras, versicherte jedoch in einem Kommentar (Chollet, 2016), dass Szegedy zu dem Zeitpunkt, ein Jahr nach Veröffentlichung des Papers, die Aktivierungsfunktion vor der BatchNormalization anwendet.

In der Veröffentlichung zum Dropout (Srivastava, et al., 2014), wiederum wird gezeigt, dass der Dropout nach der Aktivierungsfunktion genutzt wird. In (Li, et al., 2018) wurde zum anderen behauptet, dass BatchNormalization und Dropout nicht zusammen verwendet werden sollten, da BatchNormalization allein, ähnliche regularisierende Effekte wie der Dropout aufweist und

somit beide zusammen das Training verschlechtern. Chen et.al. haben allerdings untersucht, dass die Kombination von BatchNormalization in Kombination mit Dropout, vor dem Training der Gewichte, zu einem besseren Ergebnis führen (Chen, et al., 2019).

Das Thema der Anordnung ist weiterhin unentschlossen und abhängig vom Anwendungsfall. Daher wurden für diese Arbeit verschiedene Anordnungen implementiert und verglichen.

Loss Funktion

Die Wahl der Loss Funktion ist, beim Training eines Modells, einer der wichtigsten Aspekte. Wie in *2.3.4* erklärt, entscheidet diese darüber wie gut die Ausgabe des Modells ist und wie weiter trainiert wird. Dabei gibt es in Keras die Möglichkeit eigene Loss Funktionen zu erstellen oder vorgegebene zu nutzen. Da sich die Loss Funktion sehr stark nach dem Anwendungsfall und Ziel des Modells richtet, gibt es in der Keras API um die 30 Loss Funktionen, welche in drei Kategorien aufgeteilt sind:

- Probabilistische Loss Funktionen, welche eine Wahrscheinlichkeitsverteilung über eine Menge an Klassen, statt einer einzigen Klasse berechnen. Diese eignen sich daher bei Klassifikationen.
- Regression beschreibt die Vorhersage stetiger Werte, weshalb sich diese Kategorie von Loss Funktionen beispielsweise für die Vorhersage von Aktien- oder Immobilienpreisen eignet.
- Die Hinge Loss Funktionen werden für die “maximum-margin” Klassifikation genutzt. Diese Art von Klassifikation ist eine binäre Klassifikation, bei der die Datenpunkte durch eine Hyperebene voneinander getrennt sind. Dabei bildet die “maximum-margin” Hyperebene die beste Separation der Datenpunkte, da sie den maximalen Abstand zu den Punkten aus den beiden Mengen hat, was dazu führt, dass die Hyperebene nicht verschoben werden muss und zu guter Generalisierung führt, (Rosset, et al., 2003).

Bei Autoencodern zur Bildrekonstruktion werden zwei beliebte Loss Funktionen verwendet, der Cross entropy und MSE Loss.

MSE Loss

Die simplere dieser beiden ist die mean squared error (deutsch: Mittlere Quadratische Abweichung). Wie es der Name schon andeutet, wird bei dieser Funktion die Differenz zwischen der Ausgabe des Modells und dem erwarteten Ergebnis genommen und quadriert. Dies wird für den gesamten Datensatz durchgeführt und der Mittelwert wird als Loss zurückgegeben

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i). \tag{12}$$

n = Anzahl Datenpunkte (Bildpunkte)
y = Label
$\hat{y}$ = Modellausgabe

Das Quadrieren führt nicht nur dazu, dass das Ergebnis nie negativ ist, sondern auch zu einer starken Gewichtung hoher Loss Ergebnisse. Während dies Ausreißer minimiert, kann jedoch ein einzelner Ausreißer das Training verlangsamen. Oftmals reicht es für ein Training aber aus, wenn wenige Ausreißer vorhanden ist, das Model aber dafür insgesamt einen kleineren Loss hat. Die übliche Aktivierungsfunktion des Ausgangs, bei der Nutzung von MSE, ist sigmoid.

Cross entropy Loss

Die Cross entropy (CE) (deutsch: Kreuzentropie) Loss Funktion baut auf einer Wahrscheinlichkeitsverteilung des Ausgangs und des erwarteten Ergebnisses auf. Umso mehr der Ausgang von der Erwartung divergiert, desto höher wird der Loss, was bei dem CE Loss zu einem schnelleren Training führt.

Da bei der CE eine Verteilung ausgegeben wird, muss sichergestellt werden, dass diese gültig ist, das heißt keine Werte größer als 1 sind und die Werte zusammen 1 ergeben. Das ist rein mit CE nicht gegeben, weshalb die softmax Aktivierungsfunktion als Standard dazu genutzt wird, welche die Verteilung in eine valide umwandelt. Diese Funktion wird auch categorical-cross entropy genannt.

Für Klassifikationsprobleme, bei denen nur eine Klasse vorkommen kann, ist dies passend. Wenn allerdings mehr als eine Klasse vorkommen kann, passt diese Loss Funktion nicht mehr. Dafür gibt es die binäre-cross entropy, bei der Klassen als Label übergeben werden können und diese als entweder richtig oder falsch erkannt werden. Daher wird diese Loss Funktion mit der sigmoid Aktivierungsfunktion gekoppelt

$$CE = -\frac{1}{n}\sum_{i=1}^{n} y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i)\,. \qquad (13)$$

n = Anzahl Datenpunkte (Bildpunkte)
y = Label
$\hat{y}$ = Modellausgabe

Weitere Hyperparameter

In diesem Abschnitt werden weitere wichtige Hyperparameter, wie die Epochen, Batch Size und Optimizer besprochen.

Epochen

Ein Modell wird eine gewisse Anzahl von Iterationen lang trainiert, auch genannt Epochen. Bei jeder Epoche wird der gesamte Trainingsdatensatz einmal vorwärts und rückwärts durch das Modell gereicht. Die Anzahl an Epochen ist wichtig, da ein Model nicht zu kurz, aber auch nicht zu lange trainieren sollte. Ein Modell was zu kurz trainiert, erreicht nicht das beste Ergebnis, was es erreichen könnte. Ein Modell, welches zu lange trainiert sieht entweder keinen Fortschritt mehr oder verliert an Generalisierung und das Training sollte gestoppt werden.

Batch Size

Die Batch Size legt die Anzahl der Daten fest, die zusammen in einem Schritt durch das Model vor und zurück gehen. Dadurch entstehen pro Epoche mehrere Train-Schritte, bis der gesamte Datensatz benutzt wurde. Eine kleinere Batch Size führt daher zu einem längeren Training, nutzt aber weniger Speicher, da zu einem Zeitpunkt weniger Daten gebraucht werden.

Optimizer

Erinnernd an Abschnitt *2.3.5*, modifizieren Optimizer die Gewichte abhängig von den vorherigen Gewichten und somit dem vorherigen Loss, sodass diese optimal trainieren. Genauer gesagt wird die Schrittweite, die der Gradient geht, abhängig davon, ob sich dessen Richtung verändert, angepasst. Das vermeidet Plateaus oder Oszillation des Gradienten und Trainingsverlaufes.

5.1.4 Szenarien

Im ersten Schritt der Arbeit wurde ein Autoencoder entwickelt, welcher ein Eingabebild rekreieren soll. Ziel dessen war, die Entwicklung eines Autoencoders zu lernen, die Wichtigkeit der verschiedenen Komponenten in Erfahrung zu bringen und erstes Herantasten an den Datensatz, mit dem gearbeitet wurde. In den vorherigen Abschnitten dieses Kapitels wurde dieser Prozess und die daraus gewonnenen Erkenntnisse niedergeschrieben. Aus den Erkenntnissen konnte ein Autoencoder entwickelt und trainiert werden, dessen Ergebnisse untersucht wurden. Abhängig davon wurden Hyperparameter des Autoencoders angepasst und dieser wurde trainiert. Somit beginnt begann der Zyklus aus Abbildung *5-1*. In dem nächsten Abschnitt werden verschiedene Szenarien vorgestellt, in denen Hyperparameter verändert wurden und deren Auswirkung auf das Ergebnis miteinander verglichen werden. Der Vergleich wird anhand der Loss-Metriken sowie der subjektiven Einschätzung der Ergebnisse evaluiert, woraus das Modell entsprechend angepasst wird. Diese Veränderung wird mit in das nächste Szenario übernommen. Alle Szenarien wurden mit einer Trainingsdatensatzgröße von 8574 und einer Validierungsdatensatzgröße von 325 trainiert.

Der Loss für Validierungsdaten wird auch als Loss der Testdaten bezeichnet, weshalb in dieser Arbeit beide verwendet werden, um das gleiche zu beschreiben.

Insgesamt werden fünf Szenarien vorgestellt:

1. Vergleich von Loss Funktionen
2. Unterschiedliche Aktivierungsfunktionen des letzten Layers
3. Tiefe des Modells verändern
4. Vergrößerung der Größe des Bottlenecks
5. Veränderung der Art des Bottlenecks

Szenario 1: Vergleich von Loss Funktionen

Wie in Abschnitt *5.1.3* erwähnt, wird üblicherweise bei Autoencodern für Bildrekonstruktion entweder der Loss MSE oder Binary-Cross Entropy verwendet. Die Überlegung hierbei ist, dass obwohl Binary CE hauptsächliche für Klassifikation verwendet wird, diese Loss Funktion trotzdem gute Ergebnisse liefern könnte, da diese unteranderem in der Keras Anleitung für Autoencoder (Chollet, 2016) und einer Veröffentlichung zur Arbeit mit Autoencodern (Creswell, et al., 2017) genutzt wurde.

Modellarchitektur

Aus den vorherigen Analysen der Hyperparameter hat sich eine Modellarchitektur als Basis ergeben, auf die die Szenarien angewendet werden und welche angepasst wird.

Layer	Ausgabeform	Funktion
INPUT	128 x 128 x 3	
CONV1	128 x 128 x 16	Conv2D 3x3 (INPUT)
POOL1	64 x 64 x 16	MaxPooling2D (CONV1)
CONV2	64 x 64 x 32	Conv2D 3x3 (POOL1)
POOL2	32 x 32 x 32	MaxPooling2D (CONV2)
CONV3	32 x 32 x 64	Conv2D 3x3 (POOL2)
POOL3	16 x 16 x 64	MaxPooling2D (CONV3)
CONV4	16 x 16 x 128	Conv2D 3x3 (POOL3)
POOL4	8 x 8 x 128	MaxPooling2D (CONV4)
Flatten	1 x 8291	
Dense	1 x 64	
Dense	1 x 8192	
Reshape	8 x 8 x 128	
CONV5	8 x 8 x 128	Conv2D 3x3 (Reshape)
UPSAMPLE1	16 x 16 x 128	UpSampling2D (CONV5)
…	…	…
OUTPUT	128 x 128 x 3	Conv2D 3x3 (…)

Tabelle 5-1 Aufbau des Modells für Bildrekonstruktion

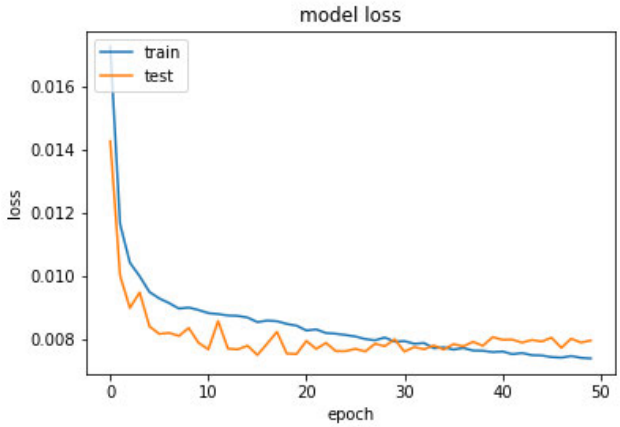
Die Layer bis zum OUTPUT spiegeln die, des Encoder Teils und erhöhen die Dimensionen des Bildes. Nach jedem Conv2D-Layer folgt ein LeakyReLU- und BatchNormalization-Layer.

Hyperparameter:

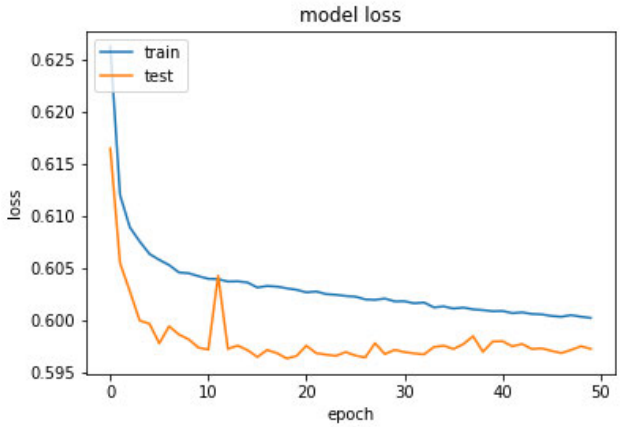
- Kernel Größe: 3x3
- Batch Size: 32
- Größe des Bottleneck: 64
- Epochen: 50
- Art des Bottleneck: Flatten + Dense
- Loss Funktion: Wird untersucht
- Optimizer: Adam mit lr=0.001
- Aktivierungsfunktion des letzten Layers: sigmoid

Ergebnisse

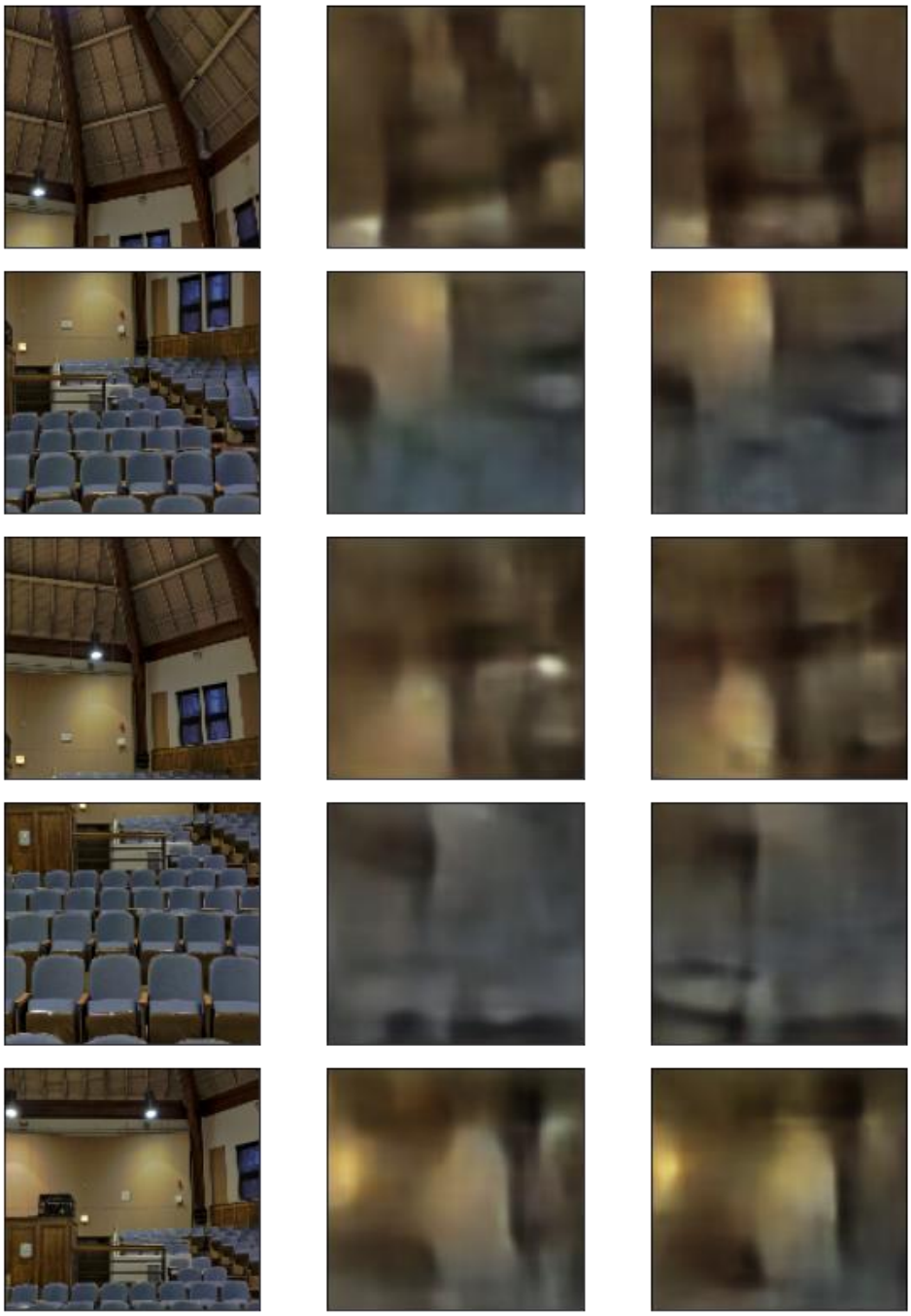
Die folgenden Ergebnisse zeigen jeweils einen Graphen, der den Verlauf der Loss-Metriken des Trainings und der Validierung beim MSE Loss und beim Binary Cross Entropy Loss. Zusätzlich sind die Modellausgaben bei beiden Loss Funktionen als Vergleich zum erwarteten Bild dargestellt.



5-2 Train und Test Loss bei MSE



5-3 Train und Test Loss bei Binary CE



5-4 Modellausgabe: Eingabebild(1.), MSE(2.), Binary CE(3.)

Evaluation

Aus dem Vergleich der Modellausgaben in Abbildung *5-4* ist zu erkennen, dass die Loss Funktionen beide ähnlich gutes Ergebnis liefern. In den Loss-Metriken (5-2 und *5-3*) ist jedoch ein großer Unterschied zu erkennen. Der Train Loss und Test Loss von Binary CE ist deutlich höher als von MSE. Die ist jedoch nicht überraschend, da der Binary CE Loss nicht auf 0 hinaus strebt, sondern auf eine Wahrscheinlichkeitsverteilung. Der MSE Loss zeigt nach einigen Epoche, dass der Train Loss geringer ist als der Test Loss, was zwar bedeutet, dass das Modell zu stark die Trainingsdaten lernt und an Generalisierung verliert, jedoch zeigt es auch, dass der

MSE Loss gut zu funktionieren scheint. Ein weiterer Aspekt, der zu beachten ist, ist die Aktivierungsfunktion des letzten Layers. Diese ist auf sigmoid gesetzt, was womöglich auf einen Vorteil für Binary CE gegenüber MSE hindeuten könnte.

Da beide Loss Funktionen jedoch gut zu funktionieren scheinen, wurde der MSE Loss für das weitere Vorgehen gewählt, da die Modellausgabe minimal besser erschien.

Szenario 2: Unterschiedliche Aktivierungsfunktionen des letzten Layer

Bei der Cross Entropy Loss Funktion, als Beispiel, ist gut zu erkennen, dass die Aktivierungsfunktion des letzten Layers eine wichtige Rolle spielt und stark von der Loss Funktion abhängt. Die Funktionen Linear und ReLU sind als passend etabliert, wenn die MSE Loss Funktion verwendet wird, (Ronaghan, 2018). Die lineare Funktion gibt einen Wert in der Reichweite minus Unendlich bis plus Unendlich, während die ReLU Funktion eine Reichweite von 0 bis plus Unendlich haben kann. Die sigmoid Funktion wird in der Regel in Kombination mit Klassifikation genutzt, da diese aber Werte in einem Bereich von 0 bis 1 liefert, wurde die für dieses Szenario mituntersucht.

Modellarchitektur

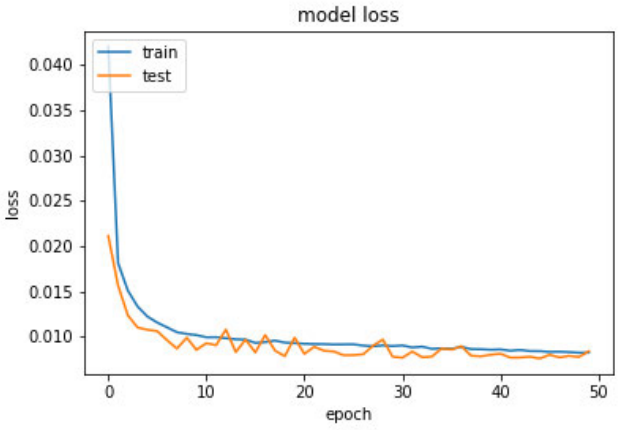
Die Modellarchitektur wurde aus dem Ergebnis von Szenario 1 übernommen und es wird die MSE Loss Funktion verwendet.

Hyperparameter:

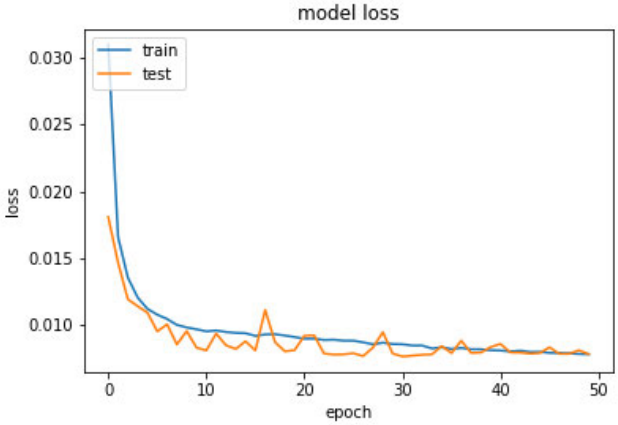
- Kernel Größe: 3x3
- Batch Size: 32
- Größe des Bottleneck: 64
- Epochen: 50
- Art des Bottleneck: Flatten + Dense
- Loss Funktion: MSE
- Optimizer: Adam mit lr=0.001
- Aktivierungsfunktion des letzten Layers: Wird untersucht

Ergebnisse

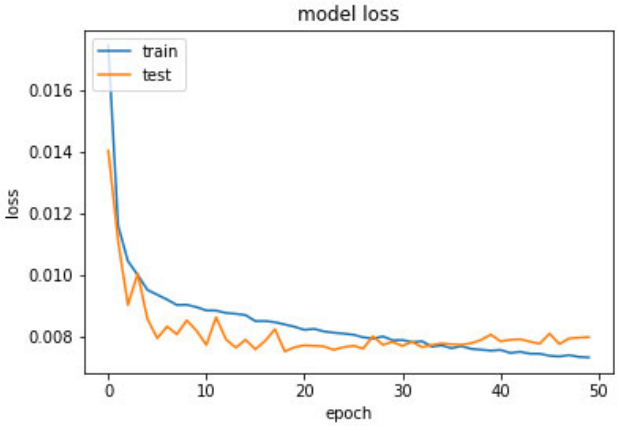
Im Folgenden sind die Ergebnisse dieses Szenarios dargestellt. Dazu ist jeweils ein Graph mit dem Train- und Validierungsloss zu sehen sowie die Ausgaben des Modells im Vergleich zum Erwartungswert.



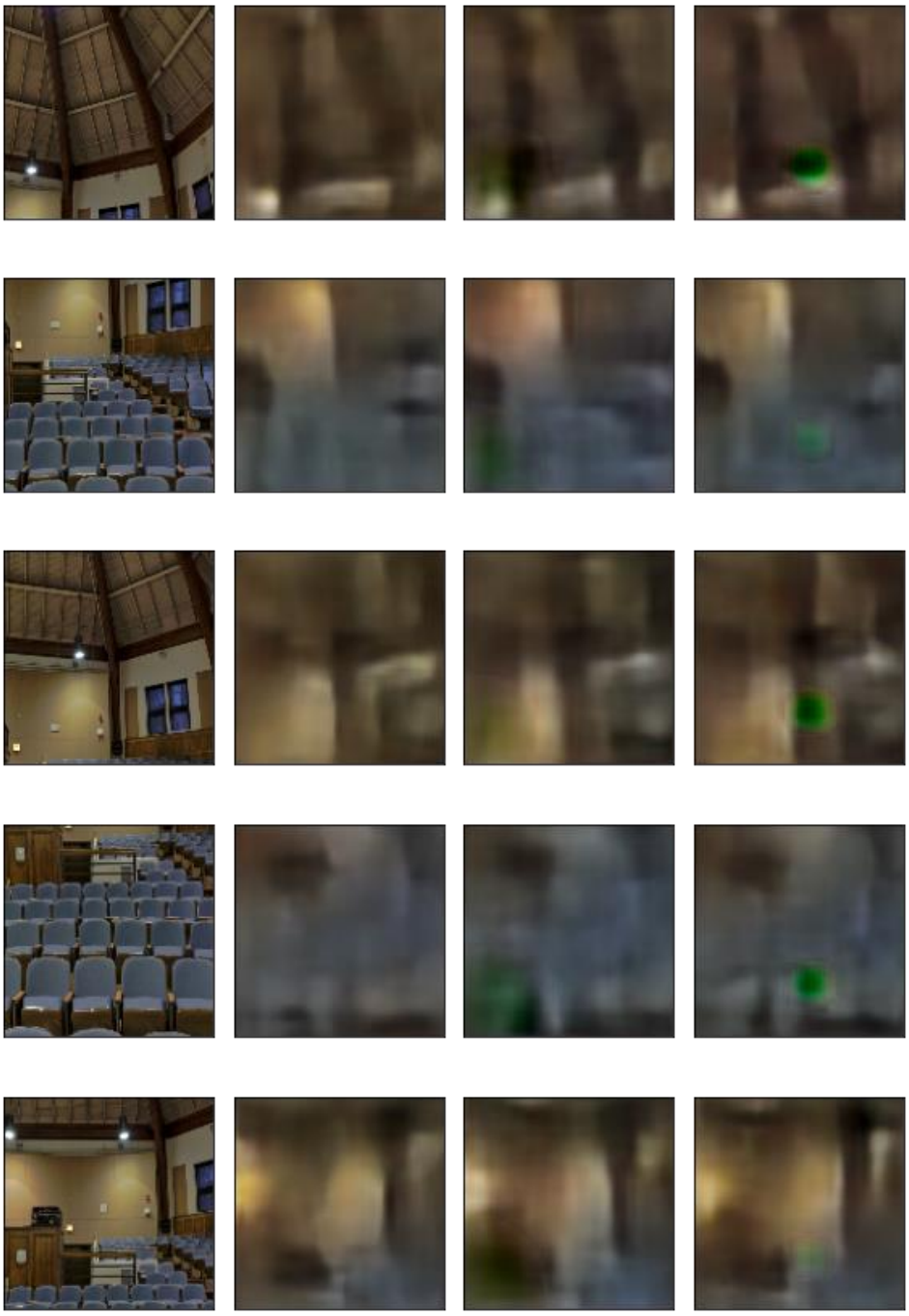
5-5 Train und Test Loss für linear



5-6 Train und Test Loss für ReLU



5-7 Train und Test Loss für sigmoid



5-8 Modelloutput. Eingabe(1.), Linear(2.), ReLU(3.), sigmoid(4.)

Evaluation

Aus den Loss-Metriken in Abbildungen *5-5*, *5-6* und *5-7* ist zu erkennen, dass alle drei Aktivierungsfunktionen ein sehr ähnliches Ergebnis zeigen. Der einzige Unterschied liegt bei der Funktion sigmoid, im Vergleich zu den anderen beiden, da dort der Train Loss, zum Ende, unter dem Test Loss liegt.

Aus den Modellausgaben (5-8) jedoch sieht man einen Unterschied. Die Funktionen ReLU und sigmoid zeigen beide einen Grünton in den Ausgaben, welcher nicht Teil des Eingabebildes ist. Von der allgemein Qualität der Ausgaben sind sie sich allerdings sehr ähnlich. Aufgrund dessen wurde sich, durch den einen Unterschied in den Ausgaben, für die lineare Aktivierungsfunktion entschieden.

Szenario 3: Tiefe des Modells verändern

Der Loss aus den vorherigen Szenarien war sehr niedrig, die Ausgaben aber noch sehr ungenau. Deshalb wurden verschiedene Tiefen des Modells und dessen Auswirkung untersucht. Ein tieferes Modell soll bekanntlich stärker trainieren, dank der erhöhten Anzahl an Neuronen. Ein flaches Modell könnte bessere Ergebnisse liefern, da die Komplexität des Modells sinkt. Ob eines dieser Veränderungen ein besseres Ergebnis liefert, wird in diesem Szenario untersucht.

Modellarchitektur

Die Modellarchitektur wird aus Szenario 2 übernommen und angepasst. Für das flache Modell werden nur zwei Convolutional Layer für jeweils Encoder und Decoder genutzt, statt 4, während bei dem tiefen Modell doppelt so viele, also 8, genutzt werden. Das flache Modell hat dadurch insgesamt 4 und das tiefe Modell 16 Layer.

Layer	Ausgabeform	Funktion
INPUT	128 x 128 x 3	
CONV1	128 x 128 x 16	Conv2D 3x3 (INPUT)
POOL1	64 x 64 x 16	MaxPooling2D (CONV1)
CONV2	64 x 64 x 32	Conv2D 3x3 (POOL1)
POOL2	32 x 32 x 32	MaxPooling2D (CONV2)
Flatten	1 x 8291	
Dense	1 x 64	
Dense	1 x 8192	
Reshape	8 x 8 x 128	
CONV3	8 x 8 x 128	Conv2D 3x3 (Reshape)
UPSAMPLE1	16 x 16 x 128	UpSampling2D (CONV3)
…	…	…
OUTPUT	128 x 128 x 3	Conv2D 3x3 (…)

Tabelle 5-2 Aufbau des flachen Modells

Layer	Ausgabeform	Funktion
INPUT	128 x 128 x 3	
CONV1	128 x 128 x 16	Conv2D 3x3 (INPUT)
CONV2	128 x 128 x 16	Conv2D 3x3 (CONV1)
POOL1	64 x 64 x 16	MaxPooling2D (CONV2)
CONV3	64 x 64 x 32	Conv2D 3x3 (POOL1)
CONV4	64 x 64 x 32	Conv2D 3x3 (CONV3)
POOL2	32 x 32 x 32	MaxPooling2D (CONV4)
CONV5	32 x 32 x 64	Conv2D 3x3 (POOL2)
CONV6	32 x 32 x 64	Conv2D 3x3 (CONV5)
POOL3	16 x 16 x 64	MaxPooling2D (CONV6)
CONV7	16 x 16 x 128	Conv2D 3x3 (POOL3)
CONV8	16 x 16 x 128	Conv2D 3x3 (CONV7)
POOL4	8 x 8 x 128	MaxPooling2D (CONV8)
Flatten	1 x 8291	
Dense	1 x 64	
Dense	1 x 8192	
Reshape	8 x 8 x 128	
CONV9	8 x 8 x 128	Conv2D 3x3 (Reshape)
CONV10	8 x 8 x 128	Conv2D 3x3 (CONV9)
UPSAMPLE1	16 x 16 x 128	UpSampling2D (CONV10)
…	…	…
OUTPUT	128 x 128 x 3	Conv2D 3x3 (…)

Tabelle 5-3 Aufbau des tiefen Modells

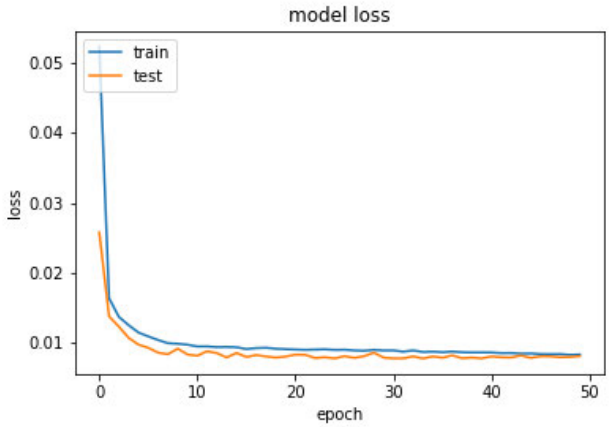
Die Layers bis zum OUTPUT spiegeln die, des Encoder Teils und erhöhen die Dimensionen des Bildes. Nach jedem Conv2D Layer folgt ein LeakyReLU und BatchNormalization Layer.

Hyperparameter:

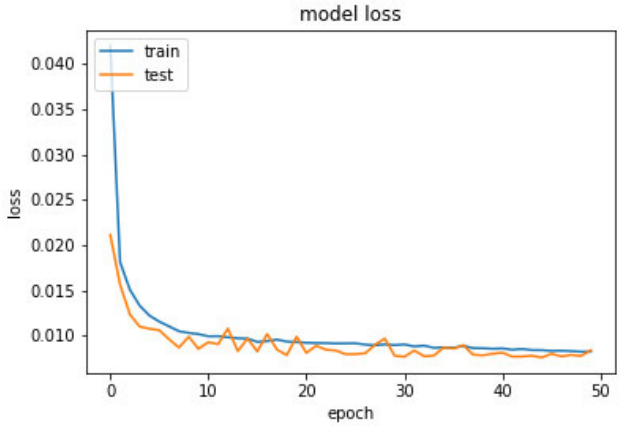
- Kernel Größe: 3x3
- Batch Size: 32
- Größe des Bottleneck: 64
- Epochen: 50
- Art des Bottleneck: Flatten + Dense
- Loss Funktion: MSE
- Optimizer: Adam mit lr=0.001
- Aktivierungsfunktion des letzten Layers: linear

Ergebnisse

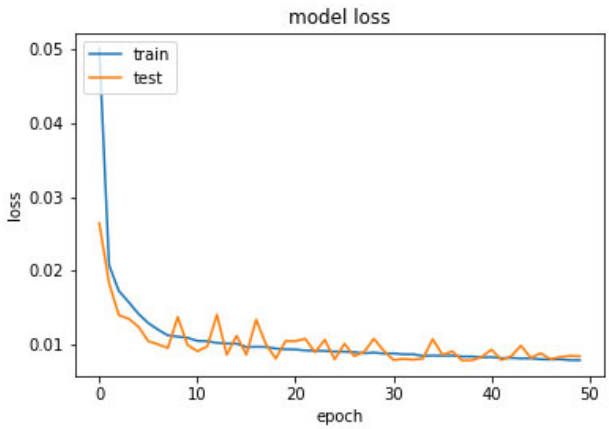
Die Ergebnisse sind in Form eines Graphen zu jeder Modellarchitektur, welcher den Train und Test Loss zeigt, dargestellt. Außerdem werden die Modellausgaben miteinander verglichen.



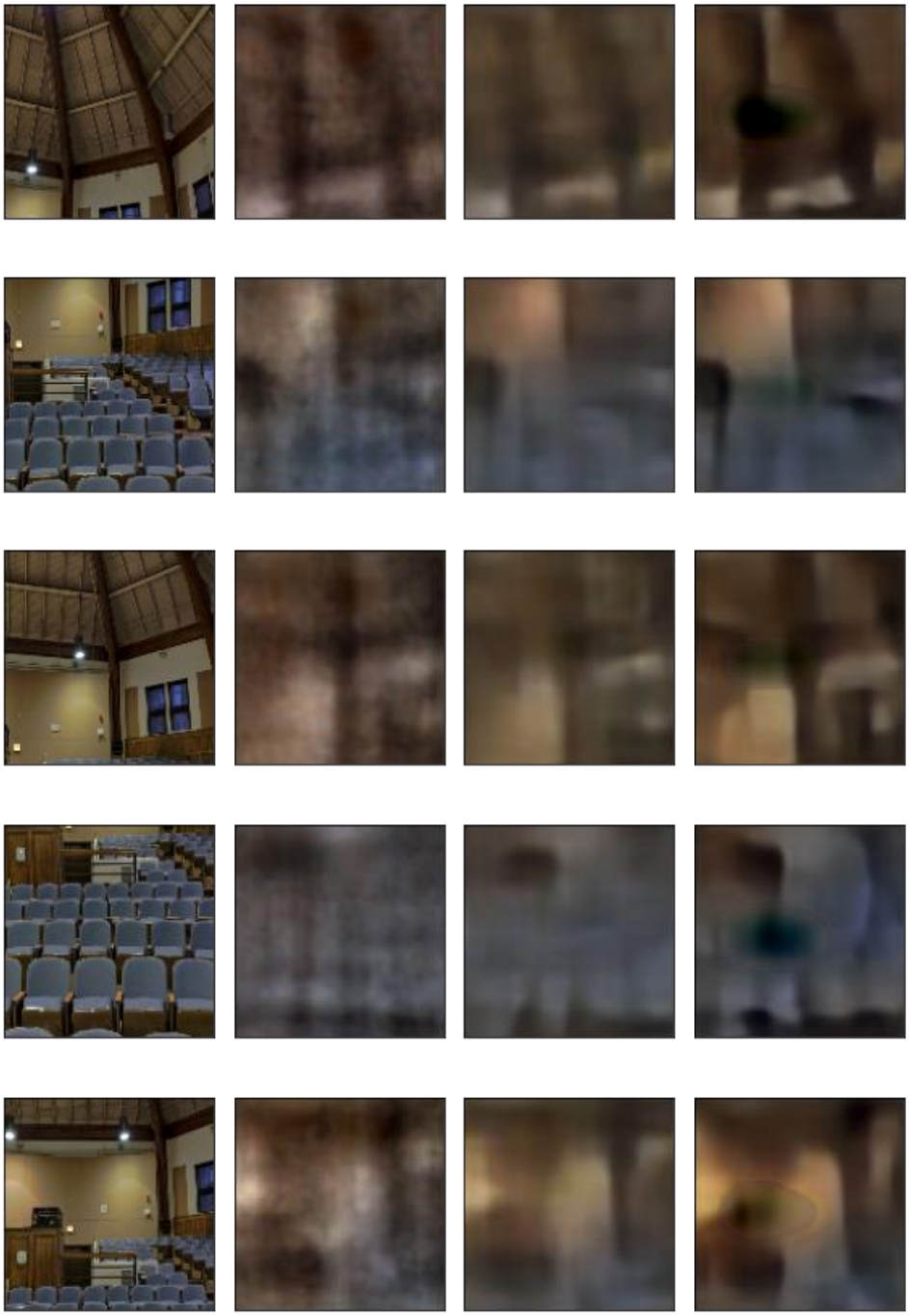
5-9 Train und Test Loss für das flache Modell



5-10 Train und Test Loss für das Modell aus Szenario 2



5-11 Train und Test Loss für das tiefe Modell



5-12 Modellausgabe. Eingabebild(1.), flaches Modell(2.), Modell aus Szenario 2(3.), tiefes Modell(4.)

Evaluation

Alle drei Tiefen des Modells zeigen ähnliche Loss-Metriken auf Abbildungen *5-9*, und *5-11*. Das flache Modell zeigt einen kleinen Vorteil dadurch, dass der Test Loss unter dem Train Loss liegt. Dagegen zeigt das tiefe Modell einen geringeren Loss, was die Hypothese unterstützt, dass dieses etwas stärker trainiert. Dadurch, dass der Test Loss aber über dem Train Loss liegt, deutet es darauf hin, dass dieses Modell sich zu stark an die Trainingsdaten richtet und and Generalisierung verliert. Die Modellausgaben (*5-12*) bestätigen dies, da die Qualität der Ausgaben des tiefen Modells klar höher ist. Aufgrund eines kleinen Rauschens bei der Ausgabe des

tiefen Modells und dessen geringe Komplexität wurde vermutet, dass das flache Modell bei längerem Training sich ebenfalls zu stark an die Trainingsdaten anpasst, was auf schlechtere Generalisierung hindeutet. Dies bestätigt Abbildung *5-13*, weshalb die Tiefe des Modells nicht verändert wurde.

5-13 Modellausgabe. Eingabebild(1.), flaches Modell(2.), Modell aus Szenario 2(3.), tiefes Modell (4.)

Szenario 4: Vergrößerung des Bottlenecks

Der Bottleneck eines Autoencoders ist der Teil, in dem die Dimensionen Feature Maps am kleinsten ist. Umso kleiner dieser Teil ist, desto größer ist die Herausforderung des Decoders diesen Teil wiederherzustellen. Bei einer sehr geringen Dimension gehen viele Details und Merkmale unter. Daher wurden die Größen 256 und 1024 untersucht und mit der vorherigen Größe 64 verglichen.

Modellarchitektur

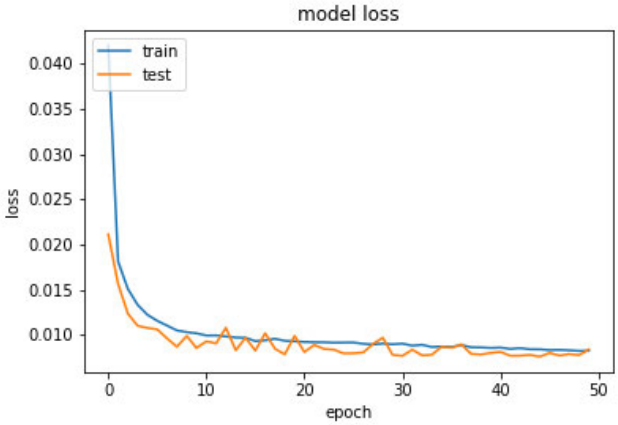
Die Modellarchitektur wird aus Szenario 2 übernommen. Der Bottleneck wird verändert und untersucht.

Hyperparameter:

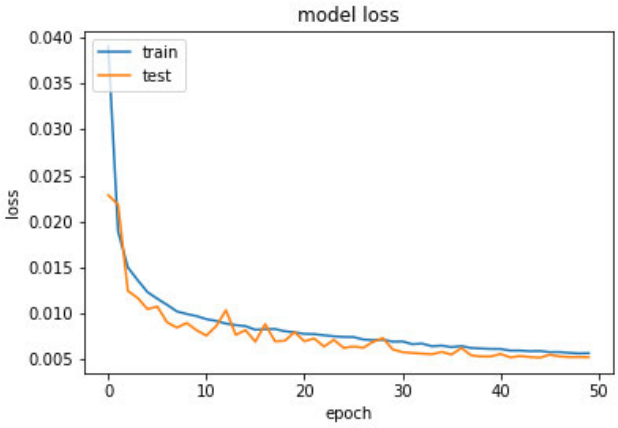
- Kernel Größe: 3x3
- Batch Size: 32
- Größe des Bottleneck: Wird untersucht
- Epochen: 50
- Art des Bottleneck: Flatten + Dense
- Loss Funktion: MSE
- Optimizer: Adam mit lr=0.001
- Aktivierungsfunktion des letzten Layers: linear

Ergebnisse

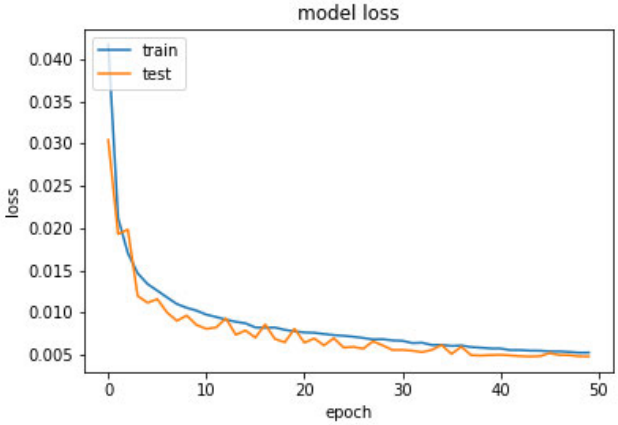
Im Folgenden sind die Ergebnisse des Szenarios dargestellt. Dazu ist jeweils ein Graph mit dem Train und Test Loss zu sehen sowie die Ausgaben des Modells im Vergleich zum Erwartungswert.



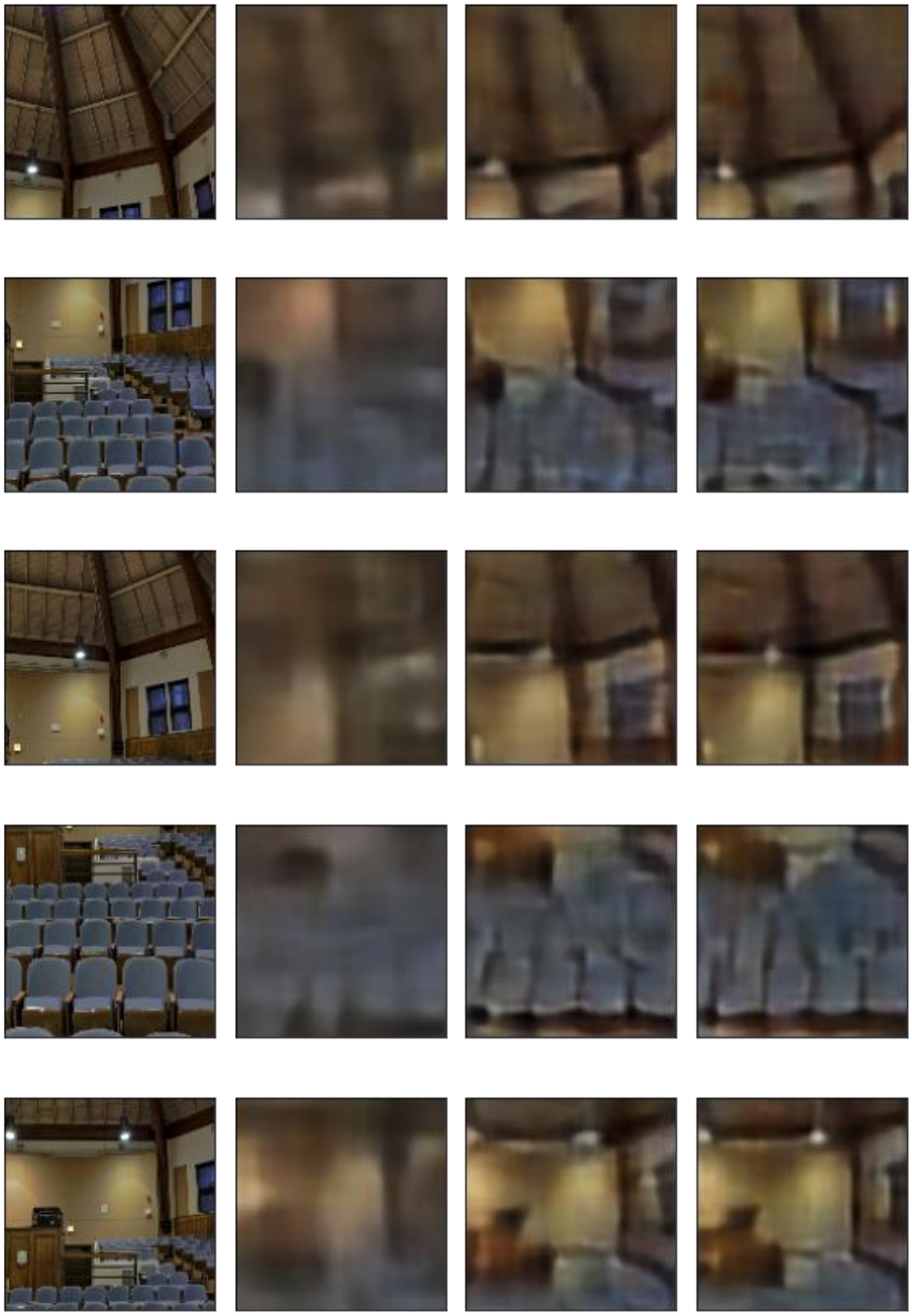
5-14 Train und Test Loss für Bottleneck = 64



5-15 Train und Test Loss für Bottleneck = 256



5-16 Train und Test Loss für Bottleneck = 1024



5-17 Modellausgabe. Eingabebild(1.), Bottleneck = 64(2.), Bottleneck = 256(2.), Bottleneck = 1024(3.)

Evaluation

In den Abbildungen *5-14*,*5-15* und *5-16* ist zu sehen, dass der Verlauf sowie das Verhältnis zwischen Train und Test Loss für die verschiedenen Größen der Bottlenecks sehr ähnlich sind. Ein großer Unterschied ist in den tatsächlichen Werten zu sehen. Der Loss bei einer Größe von 256 und 1024 ist fast doppelt so hoch, wie bei einer Größe von 64. Dies wird deutlich in den Modellausgaben in Abbildung *5-17* bestätigt.

Interessanterweise ist zwischen den Größen 256 und 1024 jedoch kaum ein Unterschied zu sehen. Ein Bottleneck, welcher größer als 256 ist, macht daher keinen Unterschied und die Größe des Bottlenecks wurde auf 256 gesetzt.

Szenario 5: Veränderung der Art des Bottlenecks

Wie in Abschnitt *2.3.6* erläutert, kann ein Convolutional-Layer mit einer Kernelgröße von 1x1 zur Reduktion von Dimensionen genutzt werden und durchaus für Autoencoder in Frage kommen. Dies wird im Folgenden untersucht und der Vergleich zwischen einem Convolutional Bottleneck und einem flachen Bottleneck wird gezeigt.

Modellarchitektur

Die Modellarchitektur verändert sich nun nur im Bottleneck Teil des Autoencoders. Daher kann der Aufbau aus Szenario 1 mit den Veränderung der Szenarien 2, 3 und 4 übernommen werden.

Layer	Ausgabeform	Funktion
…	…	…
POOL4	8 x 8 x 128	MaxPooling2D (CONV4)
CONV_BOTTLENECK	8 x 8 x 256	Conv2D 1x1 (POOL4)
CONV5	8 x 8 x 128	Conv2D 3x3 (CONV_BOTTLENECK)
…	…	…
OUTPUT	128 x 128 x 3	Conv2D 3x3 (…)

Tabelle 5-4 Modellaufbau mit verändertem Bottleneck

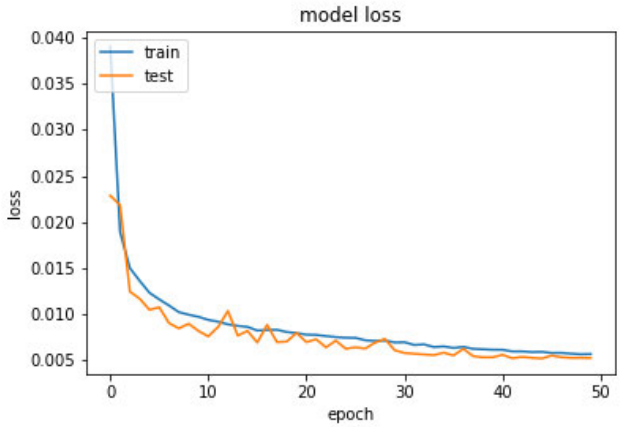
Nach dem Convolutional Bottleneck Layer folgt direkt der Convolutional Layer des Decoders.

Hyperparameter:

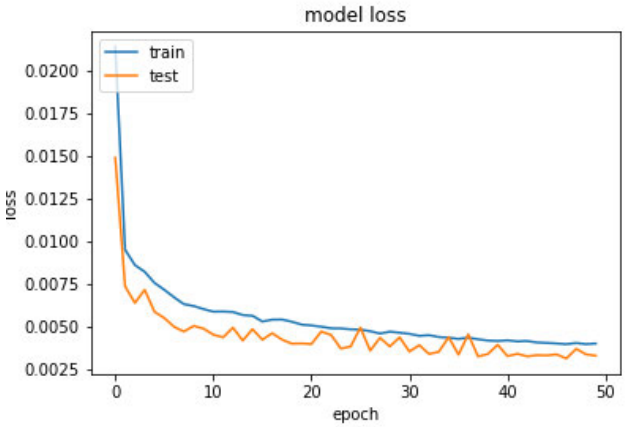
- Kernel Größe: 3x3
- Batch Size: 32
- Größe des Bottleneck: 256
- Epochen: 50
- Art des Bottleneck: Wird untersucht
- Loss Funktion: MSE
- Optimizer: Adam mit lr=0.001
- Aktivierungsfunktion des letzten Layers: linear

Ergebnisse

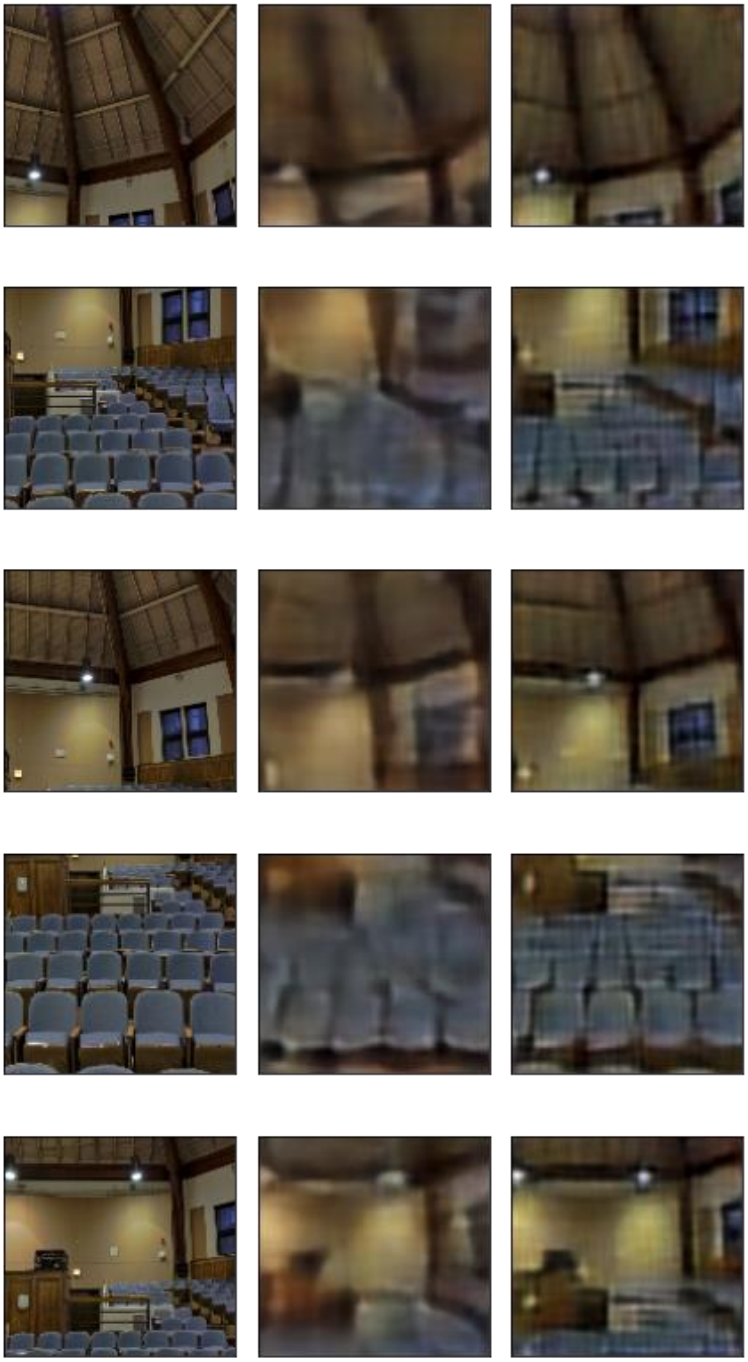
Die nächsten Abbildungen zeigen die Ergebnisse dieses Szenarios als Graphen, die den Train und Test Loss für einen flachen und einen Convolutional Bottleneck abbilden, sowie einen Vergleich anhand der Modellausgabe.



5-18 Train und Test Loss für flachen Bottleneck



5-19 Train und Test Loss für Convolutional Bottleneck



5-20 Modellausgabe. Eingabebild(1.), flacher Bottleneck(2.), Convolutional Bottleneck(3.)

Evaluation

Aus den Loss-Metriken in den Abbildungen *5-18* und *5-19* wird klar, dass der Convolutional Bottleneck etwas besser abgeschnitten hat. In den Modellausgaben in Abbildung *5-20* ist allerdings eine deutliche Besserung der Qualität, gegenüber des flachen Bottlenecks, zu erkennen. Daher hat sich bestätigt, dass ein Convolutional Bottleneck durchaus eine bessere Dimensionsreduktion durchführen kann.

Fazit zur Bildrekonstruktion

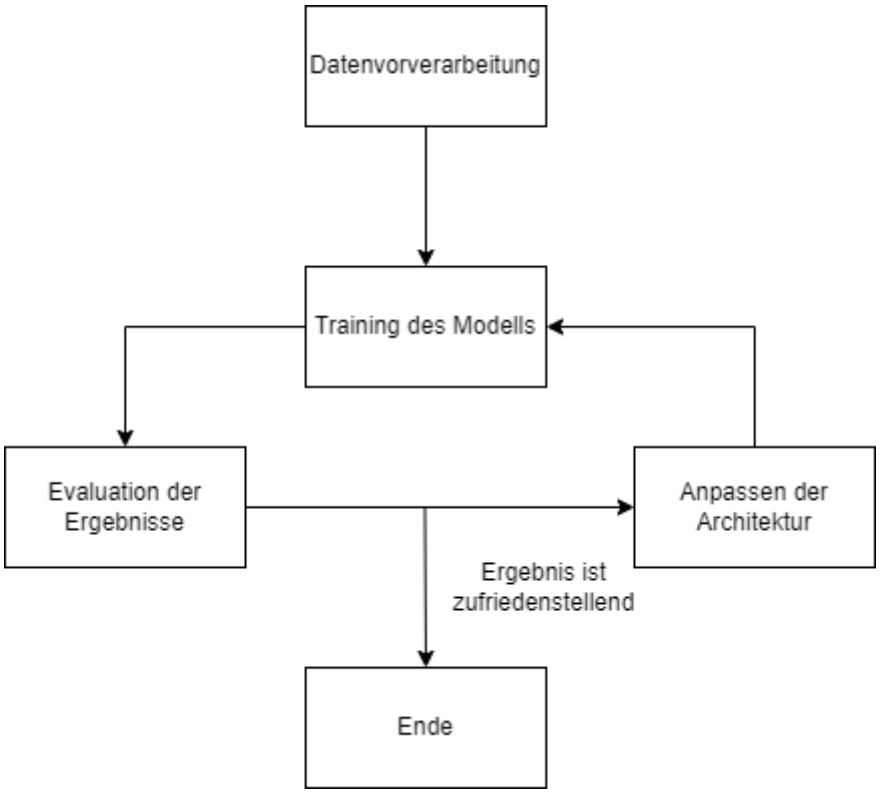
In diesem Kapitel wurde ein Autoencoder für die Bildrekonstruktion entwickelt, verschiedene Möglichkeiten untersucht und erwähnenswerte Szenarien vorgestellt.
Aus den Szenarien wurden viele Erkenntnisse über die Entwicklung eines Autoencoders gewonnen. Vermeintlich kleine Dinge, wie die Größe des Bottlenecks oder die Aktivierungsfunktion des letzten Layers, spielen eine bedeutungsvolle Rolle. Zusätzlich wurde eine gute Vorlage für das Ziel der Arbeit, die Tiefenbildrekonstruktion, gelegt. Das Modell, was aus diesen Untersuchungen entstanden ist, kann für die Tiefenbildrekonstruktion verwendet und weiter angepasst werden. Dieses vorgehen wird im nächsten Kapitel erläutert.

5.2 Tiefenbildrekonstruktion

Der Sprung von einem Autoencoder, der die Eingabe rekreieren soll, zu einem Autoencoder, der ein Tiefenbild aus einem Foto erstellen soll, ist vor allem für das Modell kein kleiner. Dieses muss nun nämlich Strukturen von Objekten wiedererkennen, einen Unterschied in der Größe feststellen und daraus bestimmen, ob sich dieses Objekt weiter vorne oder weiter hinten befindet. In diesem Kapitel werden die Schritte der Anpassungen, sowie die Untersuchungen vorgestellt und erläutert.

5.2.1 Arbeitsablauf

Der Arbeitsablauf für diesen Anwendungsfall ist sehr ähnlich zu dem vorherigen in Kapitel *5.1.1*. Das gelernte und implementierte konnte hier angewendet werden, weshalb die Wahl und Implementierung des Modells wegfallen.



5-21 Arbeitsablauf Tiefenbildrekonstruktion

Im ersten Schritt werden die Daten vorverarbeitet. Die Eingabebilder des Modells verändern sich nicht, weshalb die Vorverarbeitung für diese gleich bleibt. Die Labeldaten sind nun Tiefenbilder und die Vorverarbeitung muss für diese angepasst werden. Im zweiten Schritt wird das zuvor implementierte Modell mit den neuen Labeldaten trainiert und das Ergebnis wird im nächsten Schritt evaluiert. Die Evaluation bezieht sich hier ebenso auf die Loss-Metriken des Trainingsdatensatzes und die des Validierungsdatensatzes und zum anderen auf die eigene subjektive Wahrnehmung der Ergebnisse. Im darauffolgenden Schritt wird das Modell anhand dieser Evaluation angepasst. Dies umfasst weiterhin jegliche Hyperparameter, wie die Art und Anzahl der Layer, Loss Funktion, den Optimierer, die Epochen, Batch Size und Größe des Bottlenecks. Im Zyklus werden die Ergebnisse dieser Veränderungen evaluiert und weiterhin Anpassungen durchgeführt, bis das bestmögliche Ergebnis erzielt wird. Nachdem für die Tiefenbildrekonstruktion eine Grundeinstellung des Modells gefunden wird, werden verschiedene Szenarien entnommen und miteinander verglichen, was in Abschnitt *5.2.4* zu finden ist.

5.2.2 Datenvorverarbeitung

Für die Tiefenbildrekonstruktion werden nun die Tiefenbilder des Datensatzes als Labeldaten verwendet. Die Eingabebilder bleiben gleich, weshalb sich die Vorverarbeitung für diese nicht verändert und aus Abschnitt *5.1.2* übernommen werden kann. Die Labels jedoch sind nun komplett andere, weshalb dort eine andere Datenvorverarbeitung durchgeführt wird, die in den folgenden Abschnitten erläutert wird. Als Vorlage der Vorverarbeitung wurde der Beitrag aus (Basu, 2021) genutzt.

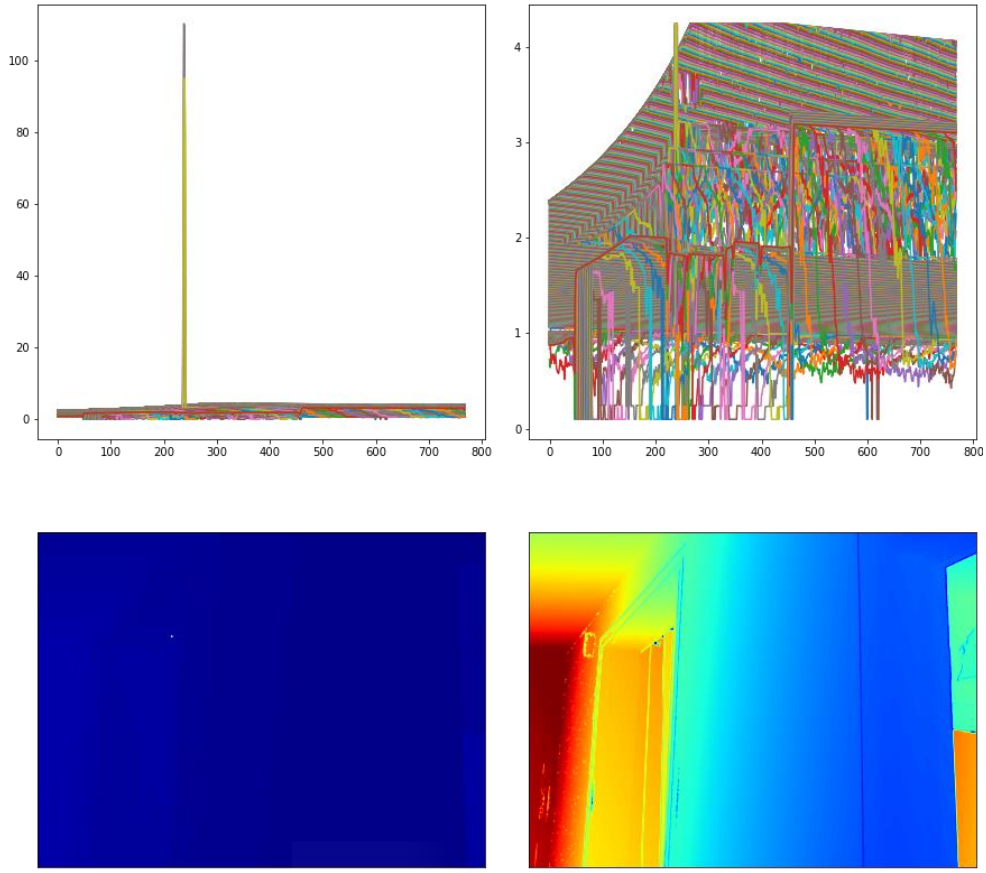
Da es in diesem Datensatz zu jedem Tiefenbild auch eine Maske gibt, muss diese nun auch berücksichtigt werden. Die Tiefenbilder werden geladen, logarithmiert, die Maske wird angewendet, Ausreißer werden entfernt und das Tiefenbild wird verkleinert.

Laden der Bilder und Masken

Genau wie die Eingabebilder müssen die Tiefenbilder und deren Maske in ein Numpy-Array geladen werden, wofür die Numpy Bibliothek verwendet wird. Beim Laden wird das Tiefenbild direkt flach geladen, damit die Maske darauf angewendet werden kann.

Ausreißer entfernen

Ausreißer in den Labeldaten können das Training stark beeinflussen, weshalb diese entfernt werden. Dafür wird zuerst die maximale Tiefe des Tiefenbildes ermittelt und mit einer Funktion der Numpy Bibliothek *clip*, werden alle Werte unter einem Minimum auf 0.1 gesetzt und alle Werte über dem Maximum werden auf die maximale Tiefe gesetzt. In Abbildung *5-22* ist die Bedeutsamkeit dieses Schritts veranschaulicht. Im Graph auf der linken Seite ist ein deutlicher Ausreißer zu erkennen, dessen Auswirkung im Tiefenbild darunter zu sehen ist. Auf der rechten Seite ist dagegen die Korrektur zu sehen.



5-22 Graph und Tiefenbild eines Tiefenbildes. Ohne clip(l.), mit clip(r.)

Logarithmieren

Das Logarithmieren hilft dabei verschiedene Größen in einen engeren Raum zu bringen. Da die später, in Abschnitt *5.2.3*, genutzte Loss Funktion mit hohen Werten einen hohen Loss liefert, bietet es sich an die Daten zu logarithmieren. Dies wurde in (Ummenhofer, et al., 2017) und (Eigen, et al., 2014) bestätigt. Dabei kann angegeben werden, welche Stellen in dem Numpy Array ignoriert werden sollen. Dazu kommt die Maske ins Spiel, die von einem Integer Array in einen Boolean Array konvertiert wurde.

Maskieren der falschen Werte

Die Maske der Tiefenbilder gibt an, an welchen Stellen die Sensormessung, bei der Datenerfassung, richtig und falsch waren. Mit einer Funktion der Numpy Bibliothek kann diese Maske auf ein Numpy Array angewendet werden.

Nach dem Maskieren werden mit *clip* ein weiteres Mal die Werte limitiert. Diesmal mit dem Logarithmus der maximalen Tiefe als Maximum.

Verkleinern der Tiefenbilder

Passend zu den Eingabebildern, werden die Tiefenbilder auf die Größe 128x128 verkleinert.

Konvertieren in einen Tensorflow Datentyp

Vor dem Konvertieren werden die Dimensionen des Tiefenbildes wieder erweitert. Anschließend wird dies, wie das Eingabebild, in einen 32 Bit Float von Tensorflow konvertiert.

5.2.3 Modellarchitektur

Das Modell aus Abschnitt *5.1* wurde übernommen, die Ergebnisse wurden evaluiert und der Zyklus begann. In diesem Abschnitt werden einzelne Erkenntnisse herausgehoben, welche das Training deutlich verbessert haben und zu der Grundeinstellung des Modells geführt haben, die für die Szenarien in Abschnitt *5.2.4* genutzt wurde.

Loss

Die zuvor verwendeten Loss Funktionen MSE und Binary Cross entropy zeigten schnell, dass ein konventioneller Loss nicht mehr ausreicht, weshalb ein benutzerdefinierter Loss genutzt wurde. In (Basu, 2021) und (Alhashim, et al., 2019) wurde die Kombination dreier Loss Funktionen genutzt, die jeweils gewichtet wurden. Beide nutzen die bekannten Loss Funktionen Structural similarity index und L1 Loss, sowie eine eigene Loss Funktion. Dafür wurde der Depth smoothness loss aus (Basu, 2021) gewählt.

Depth smoothness loss

Depth smoothness loss berechnet aus dem Gradienten Gewichte, die mit dem Gradienten der Ausgabe multipliziert werden, um die smoothness der x- und y-Ebenen der Ausgabe und daraus den depth smoothness loss zu ermitteln

$$L_{smooth} = s_x + s_y . \tag{14}$$

s ist definiert als

$$s_x = \frac{1}{n}\sum_{i=1}^{n} |g_x(\hat{y}) \cdot w_x| . \tag{15}$$

$\hat{y}$ = Modellausgabe
g_x = Gradient der x-Achse
w_x = Gewichte der x-Achse und wird berechnet durch

$$w_x = e^{\frac{1}{n}\sum_{i=1}^{n} |g_x(y)|} . \tag{16}$$

y = Labeldaten
g_x = Gradient der x-Achse

s_y und w_y werden genauso berechnet, nur für die y-Achse. Die Implementierung dieser Formel wurde aus (Basu, 2021) übernommen.

Structural similarity index (SSIM)

SSIM wurde in (Wang, et al., 2004) eingeführt und misst, grob zusammengefasst, die Ähnlichkeit zwischen zwei Bildern. Weil das menschliche Auge sehr gut in der Lage ist, solche Unterschiede zu erkennen, wurde damit ein Ansatz versucht diese Fähigkeiten zu replizieren. Dabei versucht SSIM drei wichtige Merkmale aus einem Bild zu entnehmen: Die Leuchtdichte, der Kontrast und die Struktur. Zu jedem dieser drei Merkmale gibt es noch eine zusätzliche Vergleichsfunktion zwischen dem Label und der Modellausgabe.

Die Leuchtdichte wird berechnet aus dem Mittelwert

$$\mu_x = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{17}$$

- N := Anzahl aller Pixel der Eingabe
- x := Pixel an Position i

berechnet und dessen Vergleichsfunktion ist

$$l(x,y) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1}. \tag{18}$$

C_1 ist eine Konstante und wurde auf $C_1 = (K_1L)^2$ definiert. L ist die dynamische Reichweite der Pixelwerte und K1 ist eine kleine Konstante, die deutlich kleiner als 1 sein sollte.

Der Kontrast wird berechnet mit

$$\sigma_x = \left(\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu_x)^2\right) \tag{19}$$

und hat die Vergleichsfunktion

$$c(x,y) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2}. \tag{20}$$

C_2 ist, wie C_1, eine Konstante und ist definiert als $C_2 = (K_2L_2)^2$, wobei K_2 viel kleiner als 1 sein sollte.

Letztlich wird die Struktur aus $(x - \mu_y)/\sigma_x$ und $(y - \mu_y)/\sigma_y$ berechnet.

Dessen Vergleichsfunktion lautet

$$s(x,y) = \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3}. \tag{21}$$

Dabei wird σ_{xy} definiert als

$$\sigma_{xy} = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu_x)(y_i - \mu_y). \tag{22}$$

Die drei Vergleichsfunktionen werden kombiniert und bilden die SSIM Funktion

$$SSIM(x,y) = [l(x,y)]^\alpha \cdot [c(x,y)]^\beta \cdot [s(x,y)]^\gamma. \tag{23}$$

Die drei Parameter α, β und γ müssen alle größer als 0 sein und können genutzt werden, um die Wichtigkeit der jeweiligen Formel festzulegen. Die Formeln (17) – (23) wurden aus (Wang, et al., 2004) entnommen.

Laut den Autoren von (Wang, et al., 2004) ist es von Vorteil, die Formel lokal, auf kleine Teile des Bildes, anzuwenden. Daher wird SSIM nicht über ein gesamtes Bild auf einmal ausgeführt, sondern es wird, wie bei Faltungsnetzwerken, ein Filter genutzt, der über das Bild läuft und der Mittelwert wird berechnet.

SSIM hat eine obere Schranke von 1, weshalb die Formel wie folgt lautet.

$$\frac{1}{n}\sum_{i=1}^{n} 1 - SSIM(y, \hat{y}) \tag{24}$$

y = Labeldaten
$\hat{y}$ = Modellausgabe als Batch

Der Mittelwert wird zusätzlich berechnet, da ein Batch an die Funktion über- und zurückgegeben wird.

Zu SSIM gibt es in der TensorFlow API eine Funktion:
tf.image.ssim(img1, img2, max_val, filter_size=11, filter_sigma=1.5, k1=0.01, k2=0.03).

Für *img1* und *img2* wurden jeweils Label und Ausgabe eingefügt. *Max_val* beschreibt die erlaubte Differenz zwischen den maximalen und minimalen Werten und wurde auf die Breite der Eingabe gesetzt. Der Parameter *k1* wurde, nach (Basu, 2021), auf 0.01^2 und *k2* auf 0.03^2 eingestellt. Die restlichen Parameter wurden nicht verändert.

L1 Point wise depth loss

Der L1 Loss hingegen ist relativ einfach, da dieser die absolute Differenz des Labels und der Modellausgabe nimmt

$$L_1 = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i| . \tag{25}$$

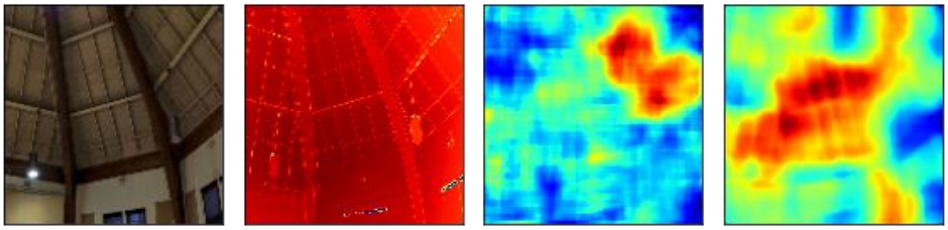
y = Labeldaten
$\hat{y}$ = Modellausgabe

Die komplette Loss Funktion sieht aus wie folgt

$$Loss = L_{Depth} \cdot w1 + SSIM \cdot w2 + L1 \cdot w3 . \tag{15}$$

Dabei bilden $w1$, $w2$ und $w3$ die jeweiligen Gewichtungen der einzelnen Loss Funktionen und wurden auf $w1 = 0{,}9$, $w2 = 0{,}85$ und $w3 = 0{,}1$ gesetzt.

Nach Implementierung und Training mit dieser Loss Funktion ist eine deutliche Verbesserung zu erkennen, was in Abbildung *5-23* gezeigt wird.



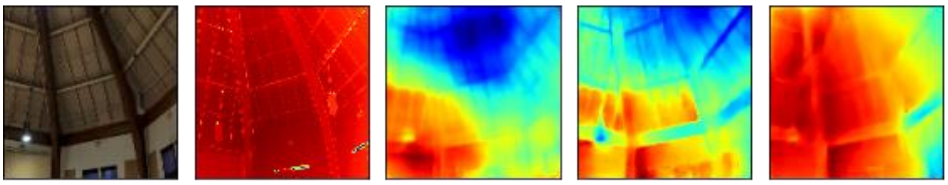
5-23 Modellausgabe. Eingabebild(1.), Label(2.), MSE(3.), Benutzerdefinierter Loss(4.)

Transfer Learning

Der Loss präsentiert eine erhebliche Besserung des Ergebnisses, zeigt jedoch noch viel Raum für Optimierung. Um das Training zu verstärken, wurde die Vergrößerung des Modells in Erwägung gezogen. Zugleich kam auch der Gedanke, ein großes vortrainiertes Modell zu nutzen. Dies hat sich in zahlreichen Anwendungsfällen [(Alhashim, et al., 2019), (Zhuang, et al., 2019) und (Weiss, et al., 2016)] als vorteilhaft herausgestellt und wurde daher auch für diesen in Betracht gezogen. Motiviert durch (Alhashim, et al., 2019) wurde ein Modell genutzt, welches auf Bildklassifikation trainiert wurde, da diese mit sehr großen Datensätzen trainierten und dadurch sehr gut darin sind, Merkmale zu extrahieren. Modelle wie VGG16/19 oder DenseNet-169/-121 werden von der Keras API zur Verfügung gestellt. Die Arbeit, die beim Training dieser Modelle geleistet wurde, kann auch mit übernommen werden, indem die Einstellung der Gewichte auf „imagenet“ gesetzt wird. Da es sich bei diesen Modellen jedoch um klassische CNN-Netzwerke handelt, müssen diese für den Gebrauch in einer Autoencoder Architektur noch angepasst werden. In der Regel werden diese Modelle angepasst, indem die letzten drei Layer nicht mit übernommen werden und für den eigenen Gebrauch angepasst werden, sodass der Teil der Klassifikation verändert wird. Das vortrainierte Modell ist dann vergleichbar mit dem Encoder Teil des Autoencoders. Der Decoder wird selbst entwickelt und erhöht die Dimensionen des Bildes wieder.

Im Vergleich zwischen VGG16 und DenseNet-169, implementiert wie in (Alhashim, et al., 2019), stellte sich DenseNet-169 als besser heraus. Dies war nicht verwunderlich, da in (Alhashim, et al., 2019) und (Vasilijevic, et al., 2019) mit dieser Architektur gute Ergebnisse erzielt wurden. Genau aus dem Grund, wird ein anderes vortrainiertes Netz, VGG16, untersucht

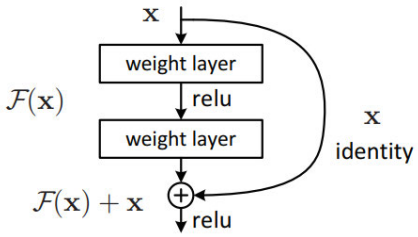
In Abbildung *5-24* wird die Ausgabe eines Modells ohne Transfer Learning, welches jedoch den gleichen Aufbau hat, wie das Modell welches Transfer Learning mit VGG16 nutzt, gezeigt. Daneben die Ausgaben mit je einem Modell, welches VGG16 und DenseNet-169 für Transfer Learning nutzt.



5-24 Modellausgabe. Eingabebild(1.), Label(2.), Ohne Transfer Learning(3.), VGG16 Transfer(4.), DenseNet-169 Transfer(5.)

Skip Connections

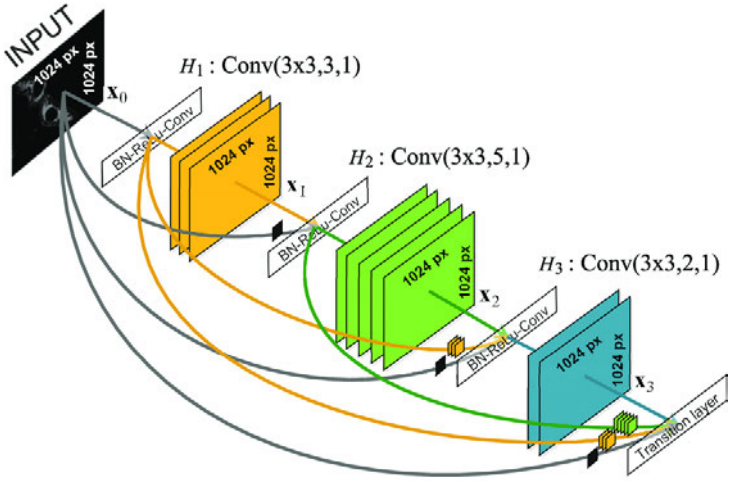
Ein tieferes Modell bedeutet zwar auf Anhieb, dass mehr gelernt wird, jedoch bringt die Tiefe auch Probleme mit sich. Wie in Kapitel 2 erläutert, kommen Probleme wie vanishing gradient, Degradation und Overfitting vor. In (He, et al., 2015) wurden Residual blocks eingeführt, die Skip Connections oder auch shortcut Connections nutzen, um das degradation problem zu lösen.



5-25 Residual Block aus *(He, et al., 2015)*

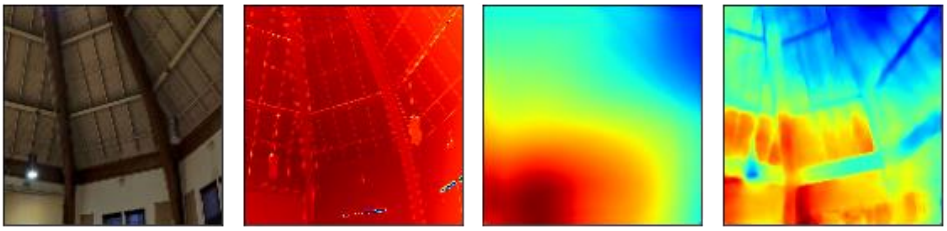
Die Ausgabe eines vorherigen Layers wird mit der Ausgabe eines späteren addiert. Dadurch ist es theoretisch möglich, dass bei solch einem Layer die Ausgabe genau gleich der Eingabe ist und somit der Layer Identiy erreichen kann. Daher bestand die Hoffnung, dass ein Modell, egal wie tief, nie schlechter werden kann, was ich in der Praxis auch bewahrheitet hat.

DenseNets (Huang, et al., 2018) nutzen auch Skip Connections, jedoch auf einer anderen Art. Bei DenseNets werden die Ausgaben von Layern nicht addiert, sondern konkateniert.



5-26 Beispiel eines Dense Blocks aus *(Škrabánek, et al., 2019)*

Dies hat zur Folge, dass die Informationen aus vorherigen Layern weitergetragen werden können und gewisse Merkmale, die zuvor extrahiert wurden, können wiederverwendet werden. Durch zusätzliche Inspiration von (Basu, 2021) und (Alhashim, et al., 2019) wurden Skip Connections in den Autoencoder eingebaut, was auch bessere Ergebnisse mit sich brachte, wie in Abbildung *5-27* zu sehen ist.



5-27 Modellausgabe. Eingabebild(1.), Label(2.), VGG16 ohne Skip Connections(3.), VGG mit Skip Connections(4.)

5.2.4 Szenarien

Das Hauptziel dieser Arbeit ist es einen Autoencoder zur Tiefenbildrekonstruktion zu entwickeln. Mit der Vorarbeit aus Abschnitt *5.1* wurde ein Autoencoder für die Bildrekonstruktion entwickelt, welcher zufriedenstellende Ergebnisse geliefert hat. In diesem Kapitel wurde dieser übernommen und für die Tiefenbildrekonstruktion angepasst. Diese Anpassungen wurden in den vorherigen Abschnitten erläutert. Der Zyklus des Arbeitsablaufes in Abbildung *5-21* begann, indem Anpassungen durchgenommen werden, evaluiert und potenziell weitere Anpassungen durchgeführt werden. Aus diesem Zyklus wurden Szenarien entnommen, welche in den folgenden Abschnitten durchgeführt werden. Dabei werden Hyperparameter des Modells verändert und die Ergebnisse werden miteinander verglichen. Zur Evaluation werden die Loss-Metriken des Trainingsdatensatzes und die des Testdatensatzes, sowie die subjektive Einschätzung der Modellausgaben, genutzt. Anders als bei der Bildrekonstruktion werden bei der Vorstellung der Modellausgaben drei Kategorien mit jeweils fünf Bildern genutzt. Diese richten sich nach den unterschiedlichen Szenen des Datensatzes und beinhalten einen Hörsaal, einen Teil einer Küche und eine Wand, vor der ein Schrank steht. Alle Szenarien wurden mit einer Trainingsdatensatzgröße von 8574 und einer Validierungsdatensatzgröße von 325 trainiert. Insgesamt wurden fünf Szenarien entnommen:

1. Unterschiedliche Aktivierungsfunktionen des letzten Layers
2. Größe des Kernels der Convolutional Layer vergrößern
3. Overfitting 1: Batch size und Größe des Bottlenecks verkleinern
4. Overfitting 2: Dropout benutzen
5. Overfitting 3: Anordnung der Layer mit Dropout anpassen

Szenario 1: Unterschiedliche Aktivierungsfunktionen des letzten Layers

In Szenario 2 aus *5.1.4* wurde untersucht, wie sich unterschiedliche Aktivierungsfunktionen des letzten Layers auf das Ergebnis auswirken, wobei die lineare Aktivierungsfunktion für das weitere Vorgehen gewählt wurde. Da sich die Labeldaten und Loss Funktion komplett verändert haben werden die drei untersuchten Aktivierungsfunktionen Linear, sigmoid und ReLU ein weiteres Mal mit den neuen Veränderungen des Modells getestet.

Modellarchitektur

Die Modellarchitektur aus Szenario 5 in *5.1.4* wurde angepasst und sieht deutlich anders aus. Den größten Teil der Veränderung spielt hierbei die Nutzung des vortrainierten Modells VGG16. Viele weitere Hyperparameter wurden jedoch übernommen.

Layer	Ausgabeform	Funktion
INPUT	128 x 128 x 3	
BLOCK1_CONV1	128 x 128 x 64	Conv2D 3x3 (INPUT)
BLOCK1_CONV2	128 x 128 x 64	Conv2D 3x3 (BLOCK1_CONV1)
BLOCK1_POOL	64 x 64 x 64	MaxPooling2D (BLOCK1_CONV2)
BLOCK2_CONV1	64 x 64 x 128	Conv2D 3x3 (BLOCK1_POOL)
BLOCK2_CONV2	64 x 64 x 128	Conv2D 3x3 (BLOCK2_CONV1)
BLOCK2_POOL	32 x 32 x 128	MaxPooling2D (BLOCK2_CONV2)
BLOCK3_CONV1	32 x 32 x 256	Conv2D 3x3 (BLOCK2_POOL)
BLOCK3_CONV2	32 x 32 x 256	Conv2D 3x3 (BLOCK3_CONV1)
BLOCK3_CONV3	32 x 32 x 256	Conv2D 3x3 (BLOCK3_CONV2)
BLOCK3_POOL	16 x 16 x 256	MaxPooling2D (BLOCK3_CONV3)
BLOCK4_CONV1	16 x 16 x 512	Conv2D 3x3 (BLOCK3_POOL)
BLOCK4_CONV2	16 x 16 x 512	Conv2D 3x3 (BLOCK4_CONV1)
BLOCK4_CONV3	16 x 16 x 512	Conv2D 3x3 (BLOCK4_CONV2)
BLOCK4_POOL	8 x 8 x 256	MaxPooling2D (BLOCK4_CONV3)
BLOCK5_CONV1	16 x 16 x 512	Conv2D 3x3 (BLOCK4_POOL)
BLOCK5_CONV2	16 x 16 x 512	Conv2D 3x3 (BLOCK5_CONV1)
BLOCK5_CONV3	16 x 16 x 512	Conv2D 3x3 (BLOCK5_CONV2)
BLOCK5_POOL	4 x 4 x 512	MaxPooling2D (BLOCK5_CONV3)
CONV_BOTTLENECK	4 x 4 x 256	Conv2D 1x1 (BLOCK5_POOL)
BLOCK1_UPSAMPLE	8 x 8 x 512	UpSampling2D (CONV_BOTTLENECK)
BLOCK1_CONCATENATE	8 x 8 x 768	Concatenate (BLOCK1_UPSAMPLE, BLOCK5_CONV1 + BLOCK5_CONV3)
BLOCK1_CONV1	8 x 8 x 512	Conv2D 3x3 (BLOCK1_CONCATENATE)
…	…	…
OUTPUT	128 x 128 x 1	Conv2D 3x3 (…)

Tabelle 5: Modellaufbau mit Transfer Learning (VGG16)

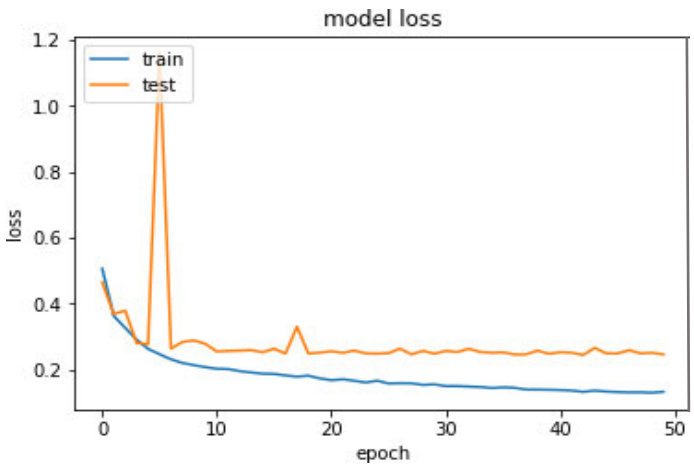
Die Layers bis zum OUTPUT spiegeln die, des Encoder Teils und erhöhen die Dimensionen des Bildes. Zusätzlich haben sie zwischen dem Upsampling2D- und Conv2D-Layer noch den Concatenate-Layer für die Skip Connections. Nach jedem Conv2D Layer folgte auch ein LeakyReLU- und BatchNormalization-Layer.

Hyperparameter:

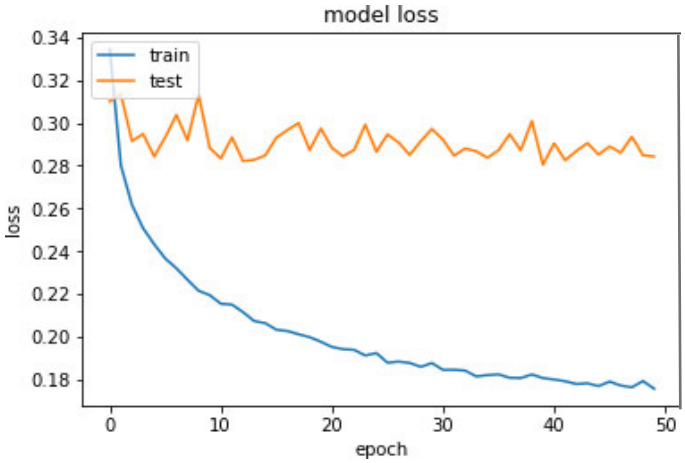
- Kernel Größe: 3x3
- Batch Size: 32
- Größe des Bottleneck: 256
- Epochen: 50
- Art des Bottleneck: Convolutional
- Loss Funktion: Benutzerdefiniert
- Optimizer: Adam mit lr=0.0001
- Aktivierungsfunktion des letzten Layers: Wird untersucht

Ergebnisse

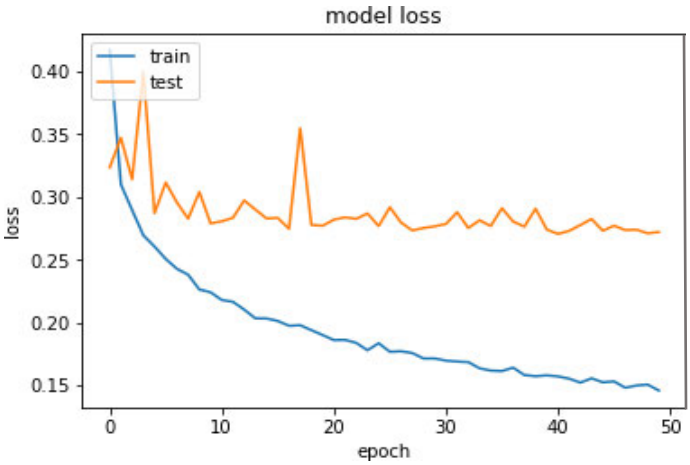
Im Folgenden sind die Ergebnisse des Szenarios zu sehen. Dazu ist jeweils ein Graph mit dem Train- und Validierungsloss zu sehen, sowie die Ausgaben des Modells im Vergleich zum Erwartungswert. Pro Szene sind 5 Ausgabebilder zu sehen.



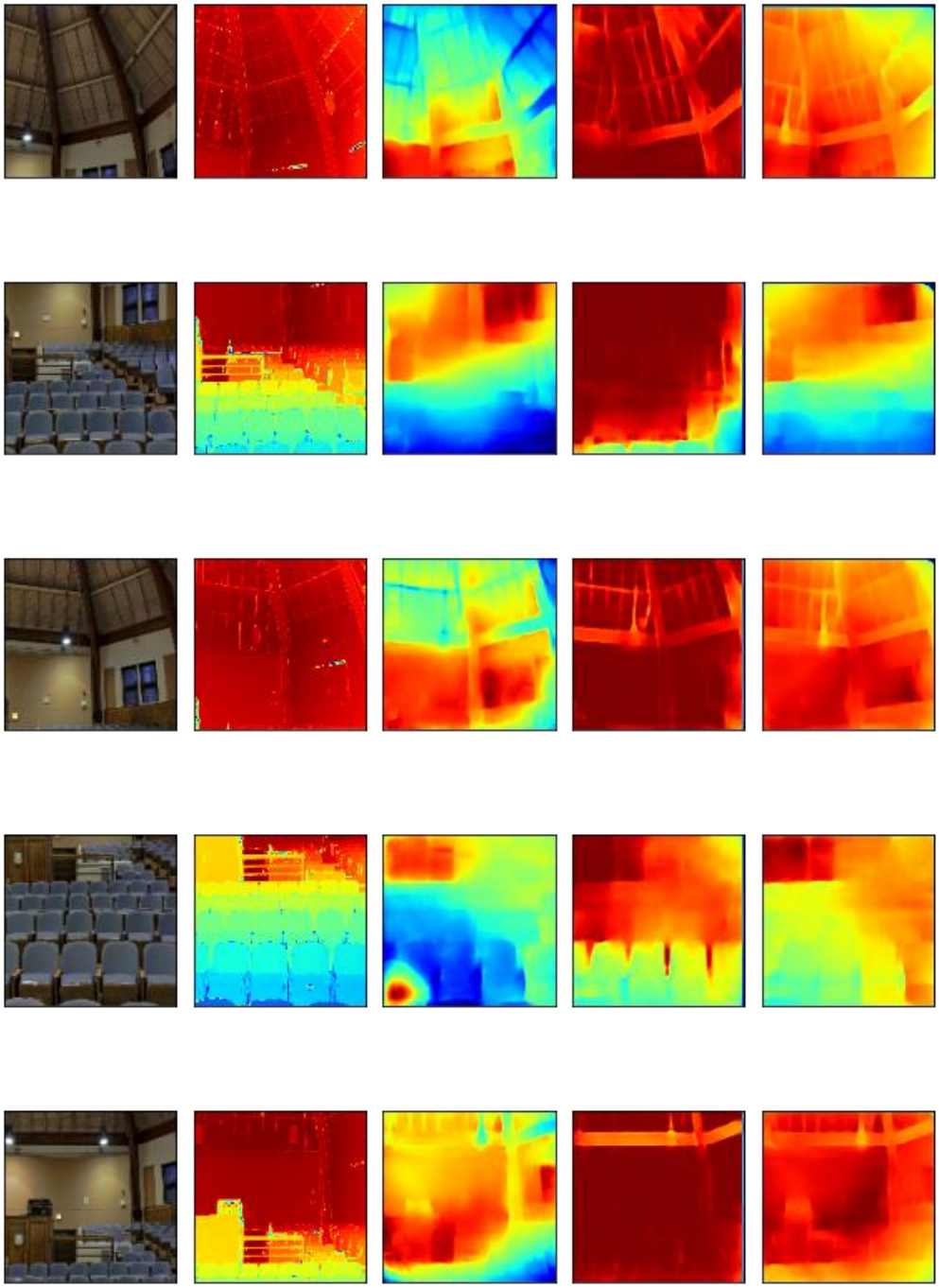
5-28 Train und Test Loss für Linear



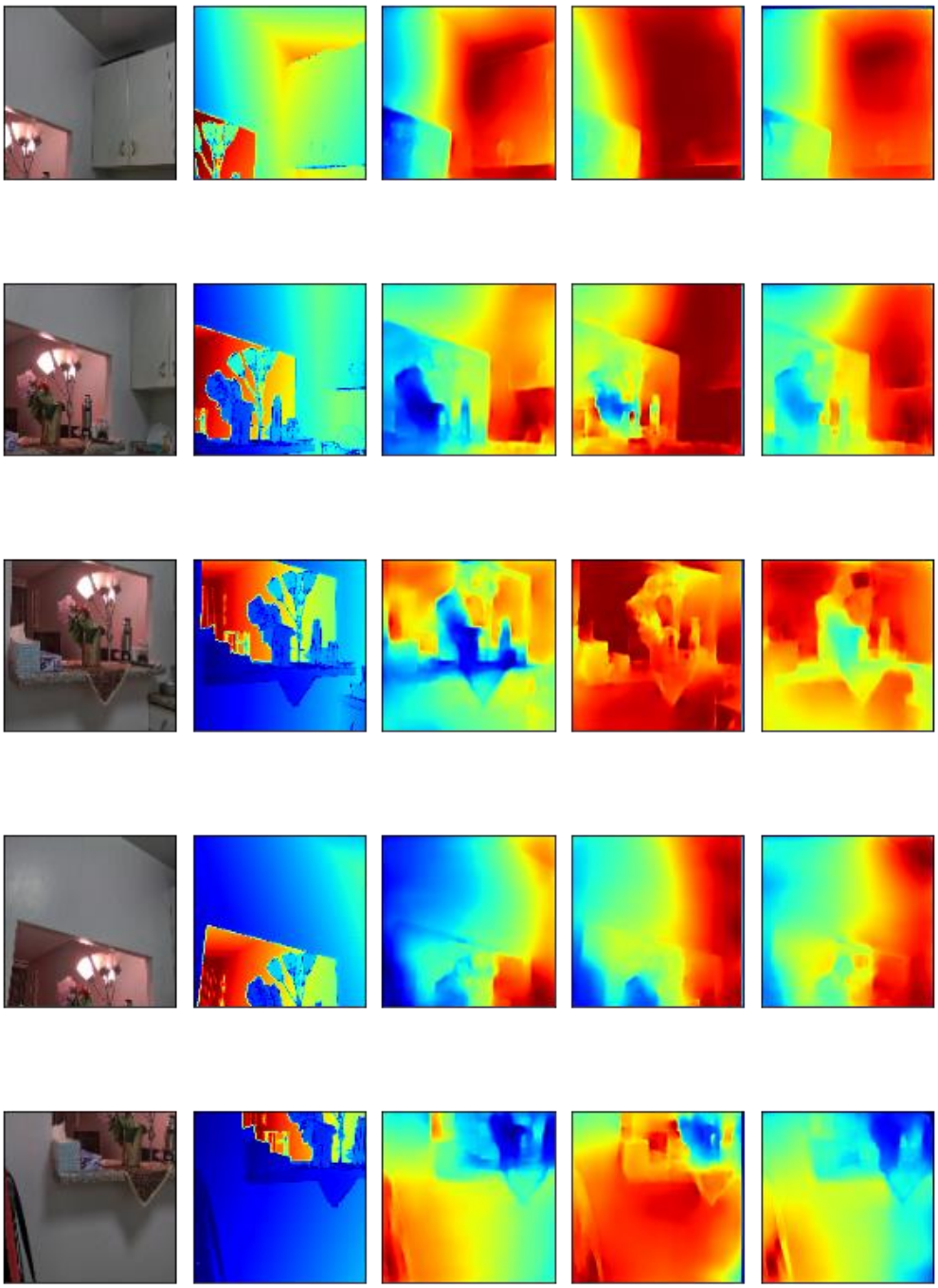
5-29 Train und Test Loss für sigmoid



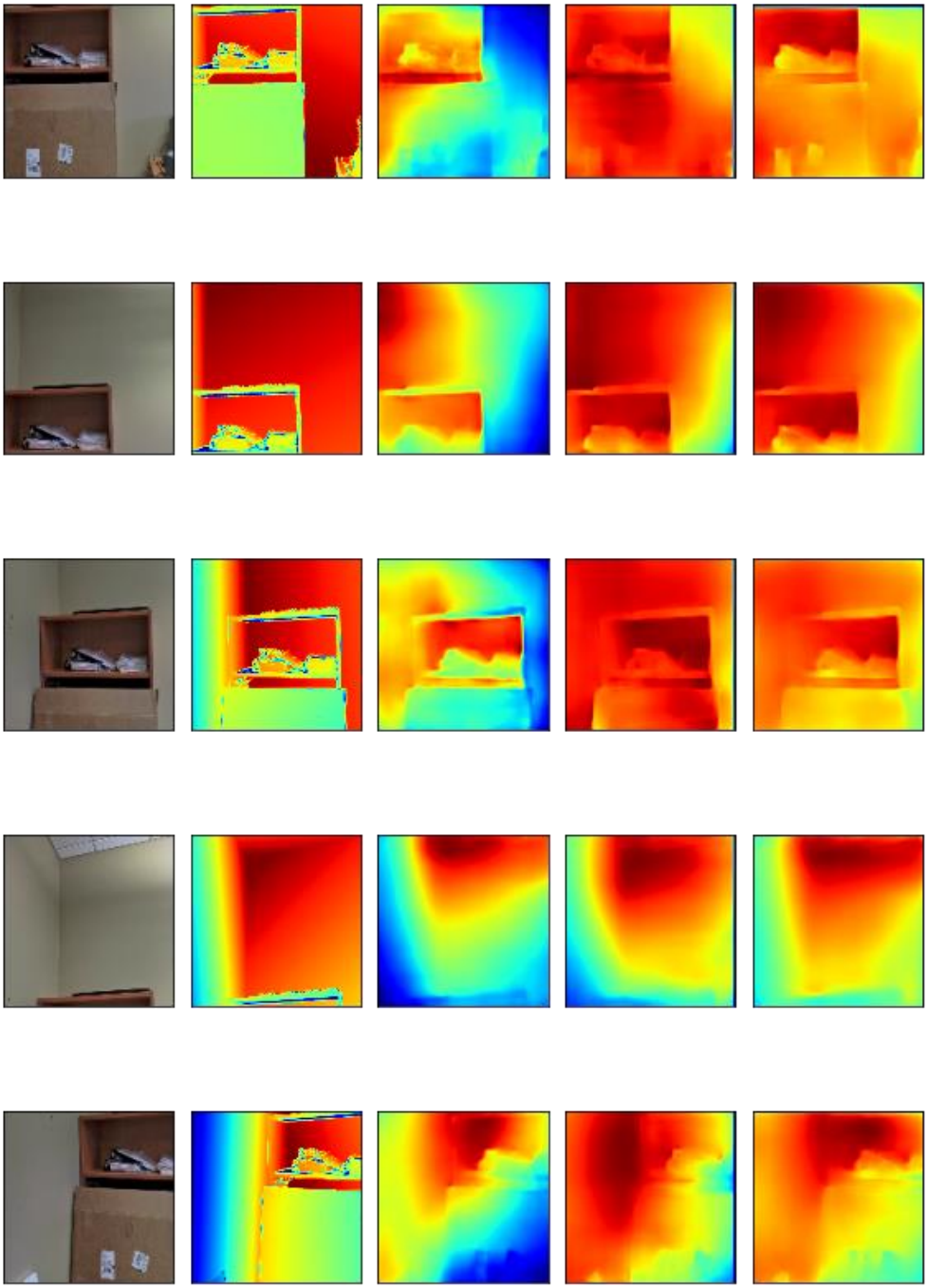
5-30 Train und Test Loss für ReLU



5-31 Modellausgabe Szene 1. Eingabebild(1.), Label(2.), Linear(3.), sigmoid(4.), ReLU(5.)



5-32 Modellausgabe Szene 2. Eingabebild(1.), Label(2.), Linear(3.), sigmoid(4.), ReLU(5.)



5-33 Modellausgabe Szene 3. Eingabebild(1.), Label(2.), Linear(3.), sigmoid(4.), ReLU(5.)

Evaluation

Rein aus den Loss-Metriken in den Abbildungen *5-28*, *5-29* und *5-30* scheint die lineare Aktivierungsfunktion am besten zu sein, da dessen Trainings und Test Loss am niedrigsten sind. Anders sieht es jedoch in den Modellausgaben in Abbildungen *5-31*, *5-32* und *5-33* aus. Grob zusammengefasst scheint es so, dass bei der linearen Funktion zu oft falsche Nahvorhersagen gemacht wurden, während, im Gegenteil dazu, bei der sigmoid Funktion zu viele falsche

Weitvorhersagen gemacht wurden. Die ReLU Funktion zeigt ein gutes Mittel zwischen Nah- und Weitvorhersagen, weshalb diese Funktion weiterverwendet wurde. Ergänzend wurde eine interessante Erkenntnis festgestellt; Die Loss-Metrik ist nicht mehr so auschlaggebend bei der Tiefenbildrekonstruktion, was in den nächsten Szenarien bestätigt wird.

Szenario 2: Größe des Kernels der Convolutional Layer erhöhen

Wie in Abschnitt *2.3.6* erwähnt, kann einen größeren Kernel bei der Faltung zu nutzen, den Vorteil haben, dass mehr Informationen bei einer Faltung gewonnen werden. Dies soll in diesem Szenario untersucht werden. Da die Größen immer in ungeraden Zahlen angegeben werden, ist die nächste Größe, nach 3x3, 5x5.

Modellarchitektur

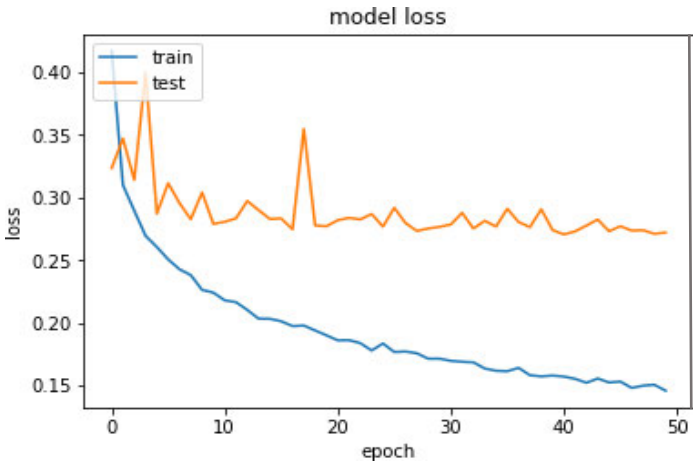
Die Modellarchitektur wird aus dem vorherigen Szenario übernommen. Die einzigen Veränderungen sind die Kernelgröße 5x5 bei den Convolutional Layer und die Aktivierungs-funktion ReLU.

Hyperparameter:

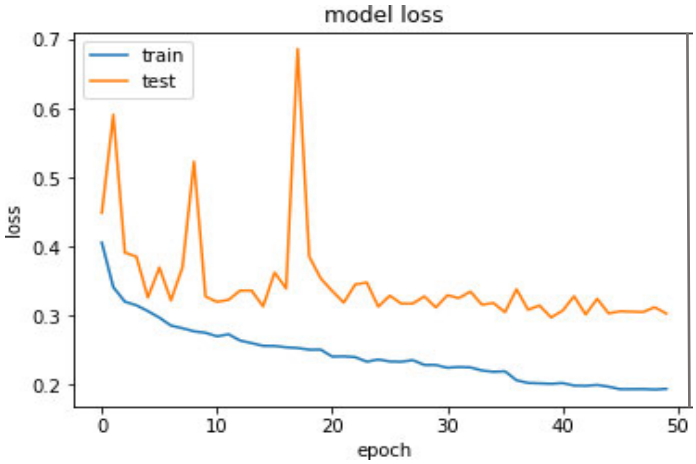
- Kernel Größe: Wird untersucht
- Batch Size: 32
- Größe des Bottleneck: 256
- Epochen: 50
- Art des Bottleneck: Convolutional
- Loss Funktion: Benutzerdefiniert
- Optimizer: Adam mit lr=0.0001
- Aktivierungsfunktion des letzten Layers: ReLU

Ergebnisse

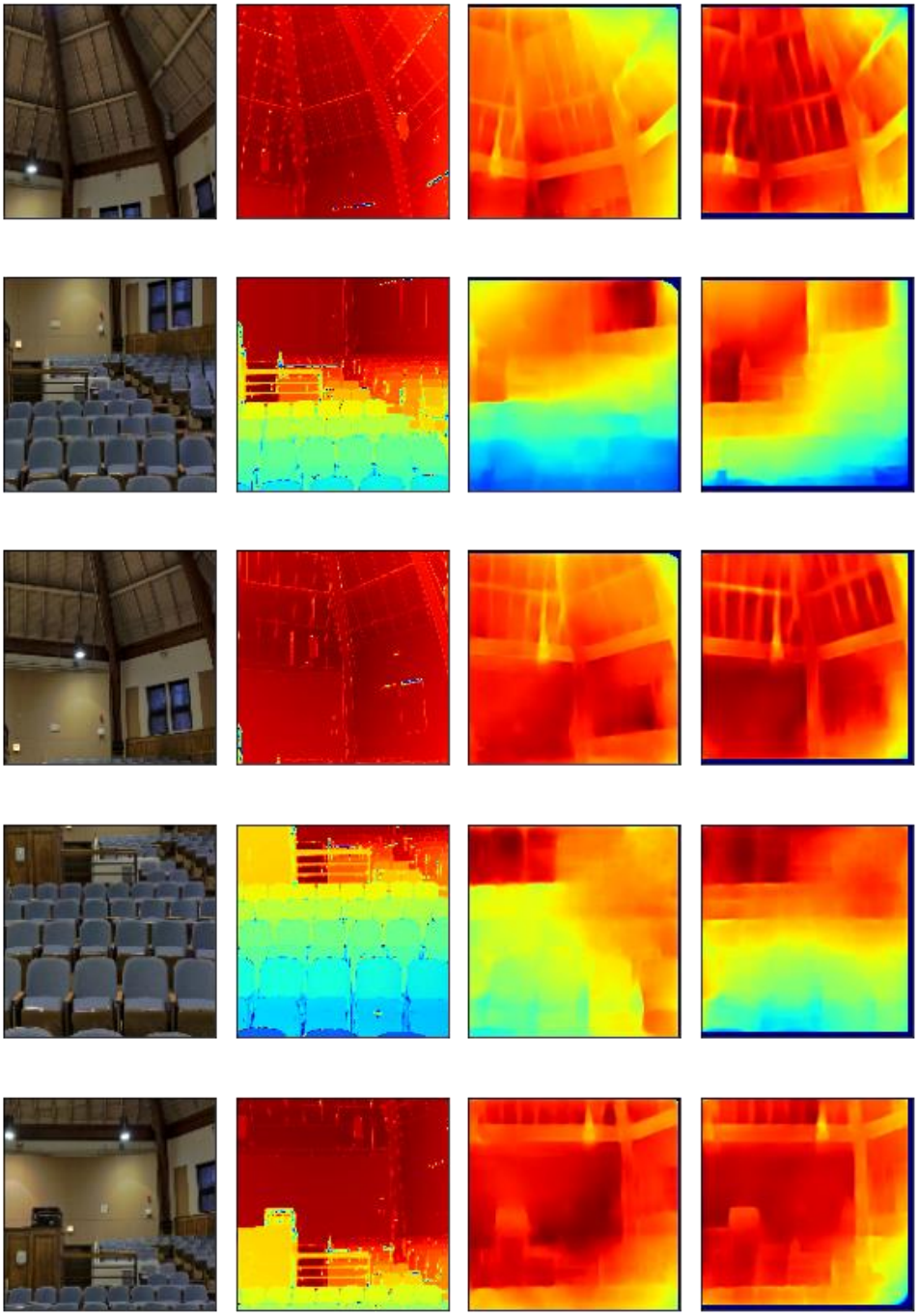
Die Ergebnisse sind in Form eines Graphen zu jeder Kernelgröße, welcher den Train und Test Loss zeigt, dargestellt. Außerdem werden die Modellausgaben der drei Szenen miteinander verglichen.



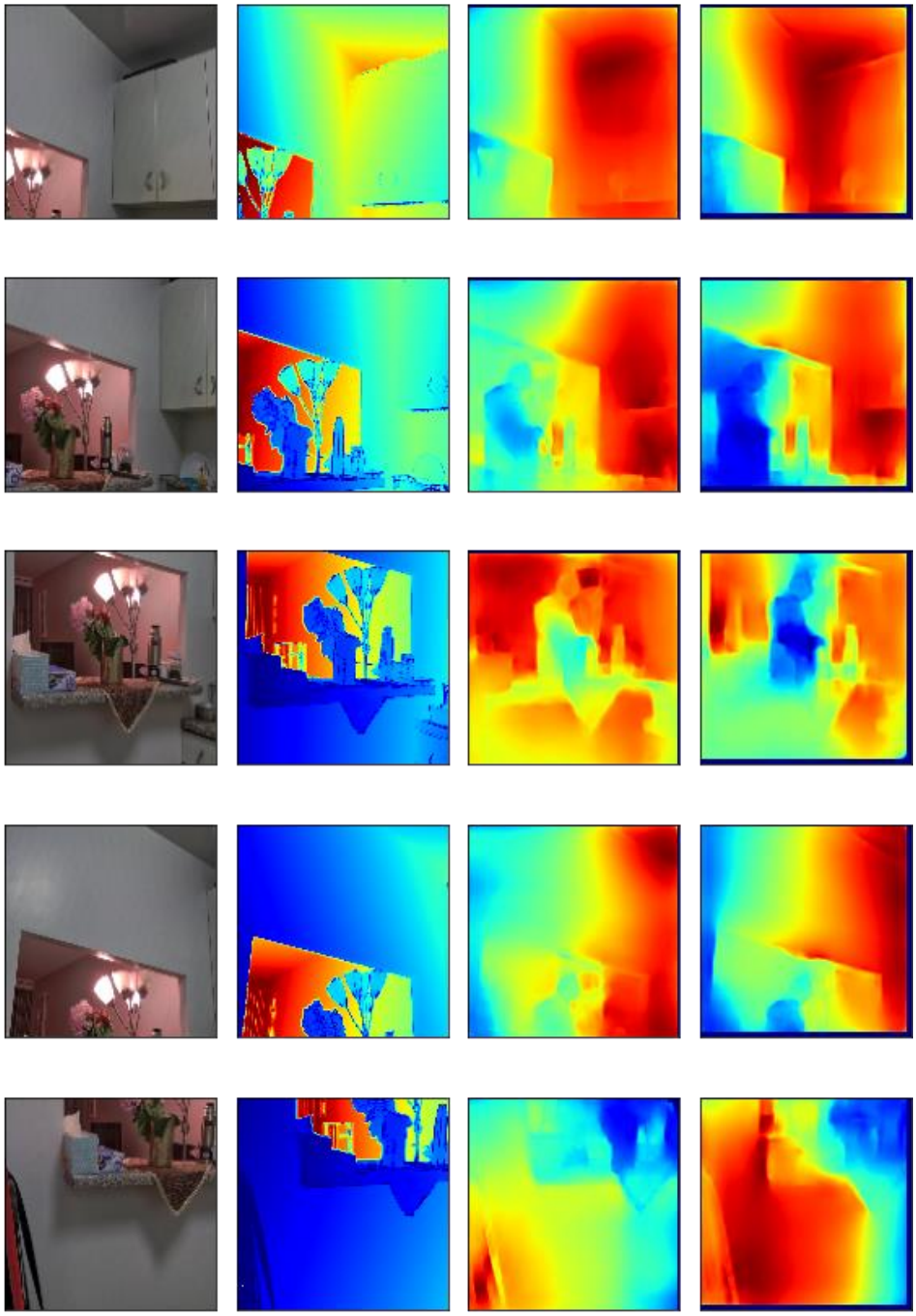
5-34 Train und Test Loss für Kernelgröße 3



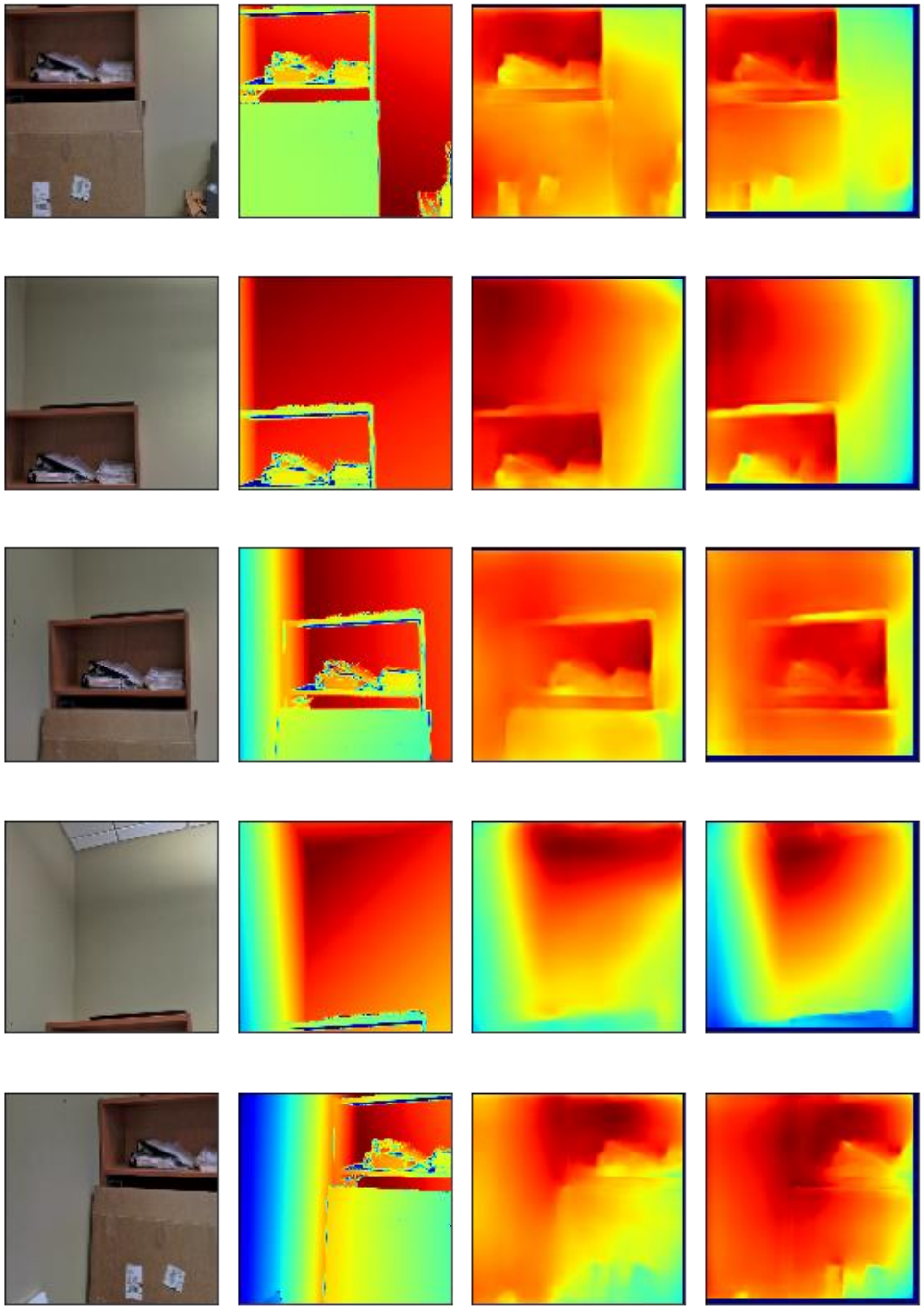
5-35 Train und Test Loss für Kernelgröße 5



5-36 Modellausgabe für Szene 1. Eingabebild(1.), Label(2.), Kernelgröße 3(3.), Kernelgröße 5(4.)



5-37 Modellausgabe für Szene 2. Eingabebild(1.), Label(2.), Kernelgröße 3(3.), Kernelgröße 5(4.)



5-38 Modellausgabe für Szene 3. Eingabebild(1.), Label(2.), Kernelgröße 3(3.), Kernelgröße 5(4.)

Evaluation

Die Loss-Metriken aus Abbildungen*5-34* und 5-35 stehen, auf den ersten Blick, zugunsten der Kernelgröße 3x3. Guckt man in die Modellausgaben in Abbildungen *5-36*, *5-37* und *5-38* sieht man, trotz ähnlicher Ergebnisse, eine etwas präzisere Vorhersage des Modells mit der Kernelgröße 5x5. Betrachtet man die Loss-Metriken ein weiteres Mal, ist zu erwähnen, dass die Metriken insgesamt, bei 3x3 geringer sind, die Diskrepanz zwischen Train und Test Loss bei 5x5

jedoch geringer ist. Wie in Abschnitt *2.3.5* erklärt, ist diese Diskrepanz oftmals ein Indiz dafür, wie gut die Generalisierung eines Modells ist. Dadurch kann ein Modell durch Einbüßen von Train Loss ein besseres Ergebnis erzielen.

Szenario 3: Overfitting: Batch size und Größe des Bottlenecks verkleinern

In den Szenarien 1 und 2 ist allgemein eine ziemlich große Diskrepanz zwischen dem Train und dem Test Loss zu beobachten. Daher wurde vermutet, dass das Modell ein wenig in das Overfitting kommt, weshalb versucht wurde, im folgenden Szenario, diesem entgegenzuwirken. Wie in Abschnitt *2.3.5* erwähnt, gibt es verschiedene Möglichkeiten, um gegen Overfitting anzugehen. Motiviert durch (Keskar, et al., 2017) wurde in einem ersten Schritt die Batch Size von 32 auf 8 reduziert. Zusätzlich wurde aus eigener Überlegung geschlossen, dass ein engerer Bottleneck, der zu einer stärkeren Dimensionsreduktion führt, das Modell dazu zwingt eine bessere Generalisierung zu erlernen. Somit wurde die Größe des Bottlenecks auf 64 reduziert.

Modellarchitektur

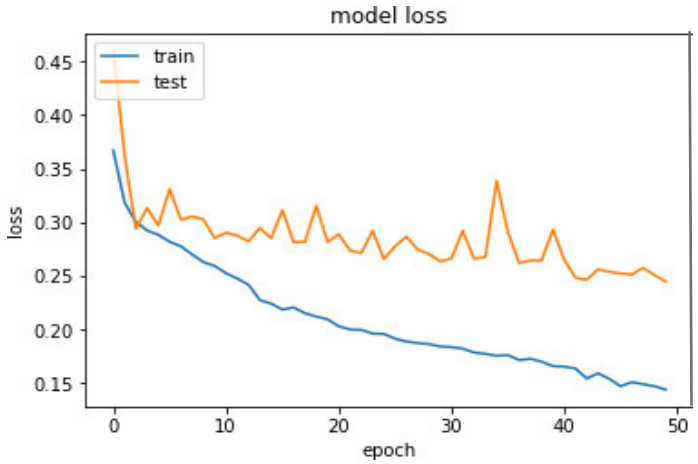
Der Aufbau des Modells kann weiter übernommen werden. Die Hyperparameter Batch Size und Größe des Bottlenecks werden verändert.

Hyperparameter:

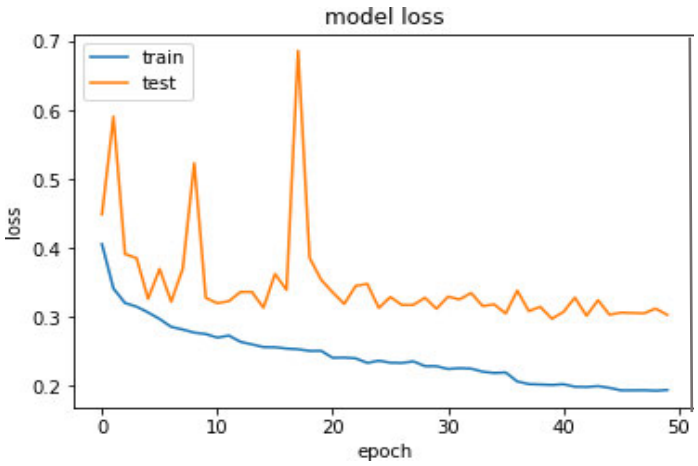
- Kernel Größe: 5x5
- Batch Size: Wird untersucht
- Größe des Bottleneck: Wird untersucht
- Epochen: 50
- Art des Bottleneck: Convolutional
- Loss Funktion: Benutzerdefiniert
- Optimizer: Adam mit lr=0.0001
- Aktivierungsfunktion des letzten Layers: ReLU

Ergebnisse

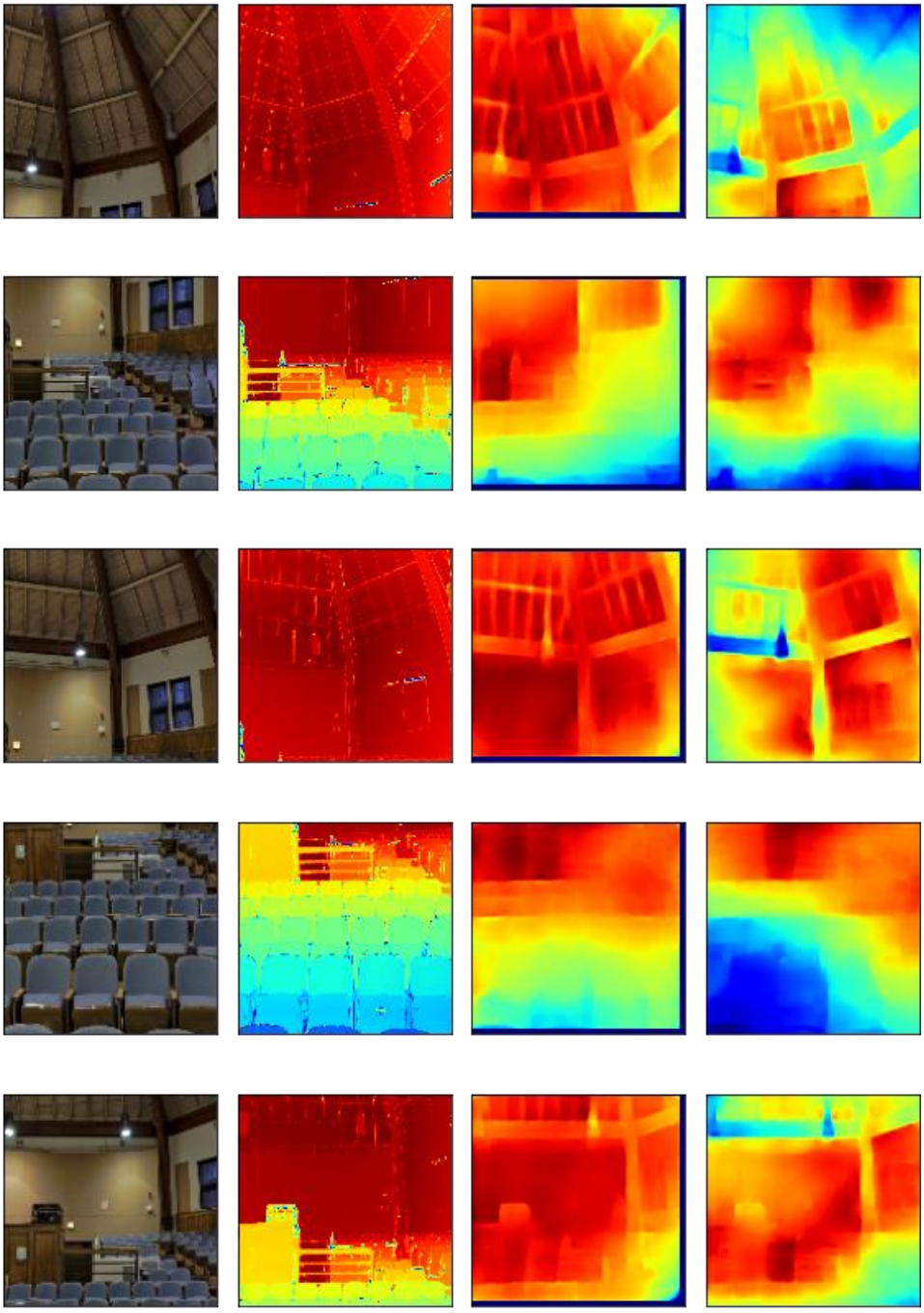
Die nachfolgenden Abbildungen zeigen einen Graphen mit dem Train und Test Loss für beide untersuchten Varianten. Zusätzlich werden die Modellausgaben für die drei Szenen im Vergleich gezeigt.



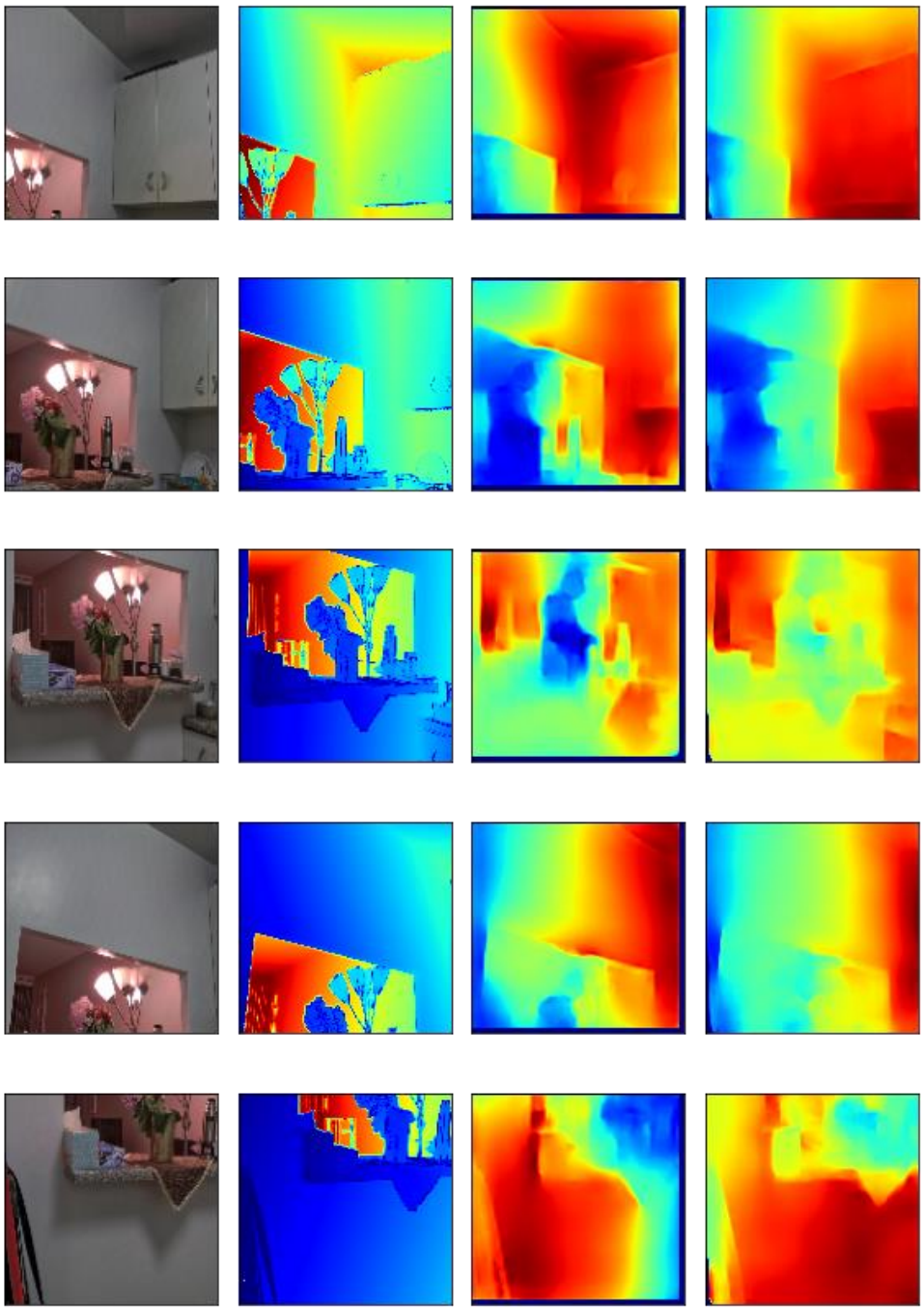
5-39 Train und Test Loss für Batch Size = 8 und Bottleneck = 64



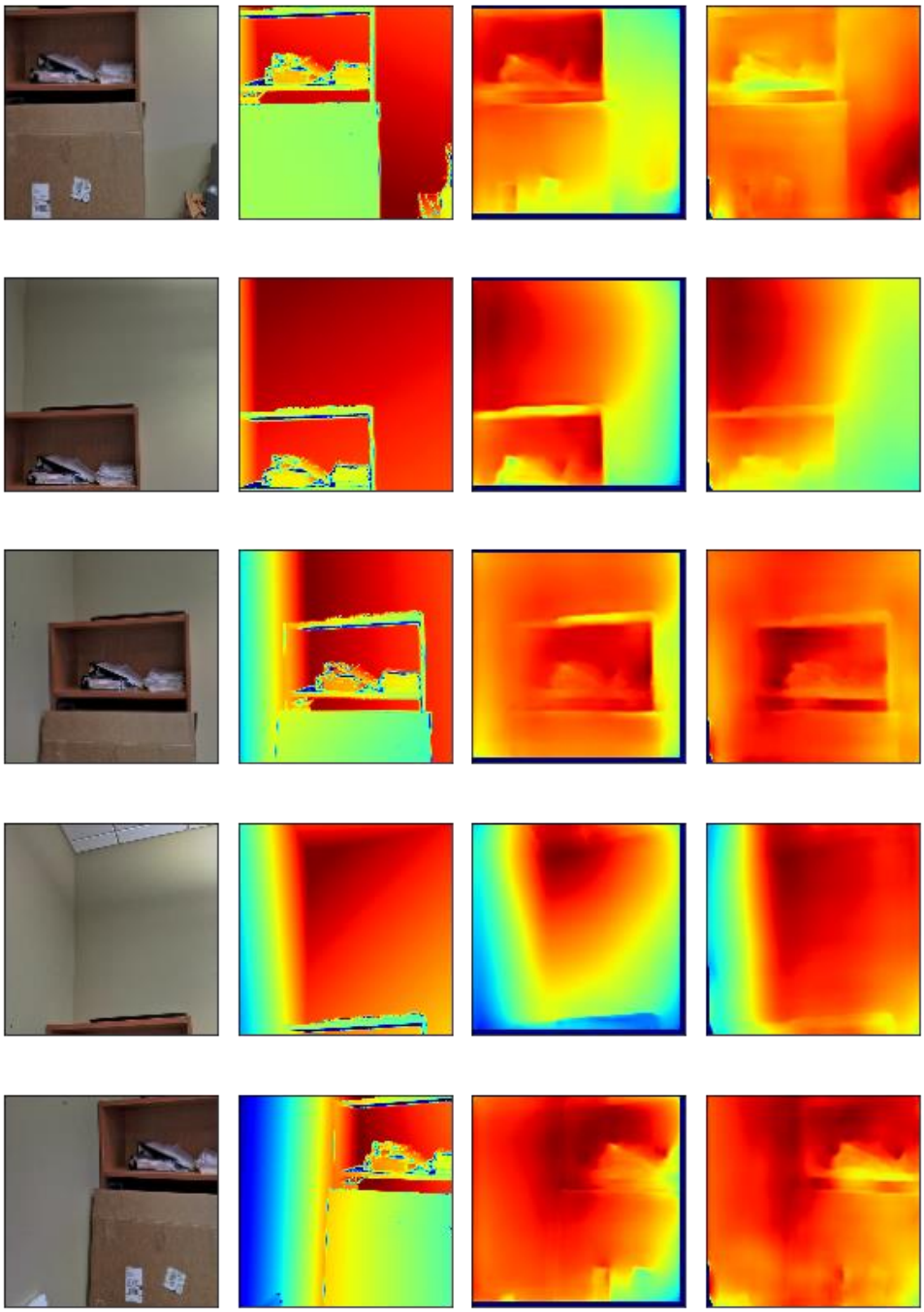
5-40 Train und Test Loss für Batch Size = 32 und Bottleneck = 256



5-41 Modellausgabe für Szene 3. Eingabebild(1.), Label(2.), Batch Size = 32 und Bottleneck = 256(3.), Batch Size = 8 und Bottleneck = 64



5-42 Modellausgabe für Szene 3. Eingabebild(1.), Label(2.), Batch Size = 32 und Bottleneck = 256(3.), Batch Size = 8 und Bottleneck = 64



5-43 Modellausgabe für Szene 3. Eingabebild(1.), Label(2.), Batch Size = 32 und Bottleneck = 256(3.), Batch Size = 8 und Bottleneck = 64

Evaluation

Ähnlich wie in Szenario 2 täuscht die Loss-Metrik in diesem Szenario. Der Train Loss und Test Loss bei kleinerer Batch Size und Bottleneckgröße ist nicht nur geringer, sondern auch die Diskrepanz zwischen diesen (Abbildungen *5-39* und *5-40*). In den Modellausgaben in Abbildungen *5-41* bis *5-43* zeigt besonders *5-41* ein deutlich besseres Ergebnis. Da keine Verbesserung festgestellt wurde, werden Batch Size und Größe des Bottlenecks nicht verkleinert und es wird ein anderer Ansatz versucht.

Szenario 4: Overfitting 2: Dropout benutzen

Eine weitere Maßnahme gegen Overfitting, ist ein Dropout zu benutzen, welcher in Abschnitt *2.3.5* erklärt wurde. Dropout ist zudem eine deutlich wirksamere Strategie, da einige Neuronen gar nicht trainieren. Obwohl, wie in Abschnitt *5.1.3* diskutiert, davon teilweise abgeraten wird Dropout und BatchNormalization zusammen zu verwenden, wurden bei den Untersuchung festgestellt, dass das Training ohne BatchNormalization gar nicht voranläuft. In diesem Szenario wird das Dropout mit einer Rate von 0.2 verwendet und mit dem Ergebnis aus Szenario 2 verglichen.

Modellarchitektur

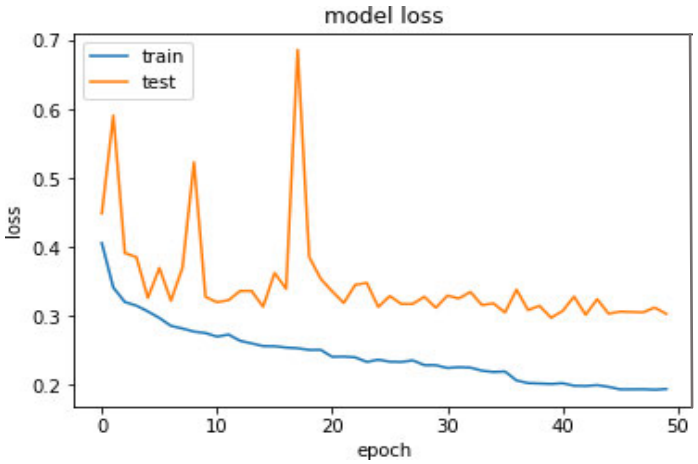
Die Architektur aus Szenario 2 wird übernommen und Dropout Layer werden hinzugefügt. Nach jedem BatchNormalization Layer wird ein Dropout Layer eingefügt, sodass die Reihenfolge Conv2D → LeakyReLU → BatchNormalization → Dropout ergibt.

Hyperparameter:

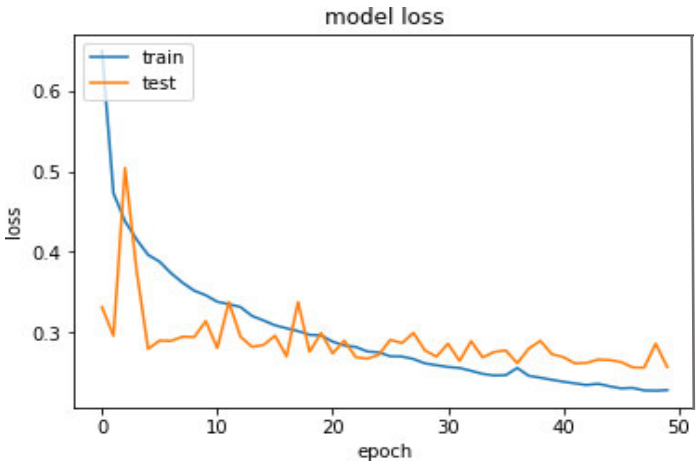
- Kernel Größe: 5x5
- Batch Size: 32
- Größe des Bottleneck: 256
- Epochen: 50
- Art des Bottlenecks: Convolutional
- Loss Funktion: Benutzerdefiniert
- Optimizer: Adam mit lr=0.0001
- Aktivierungsfunktion des letzten Layers: ReLU

Ergebnisse

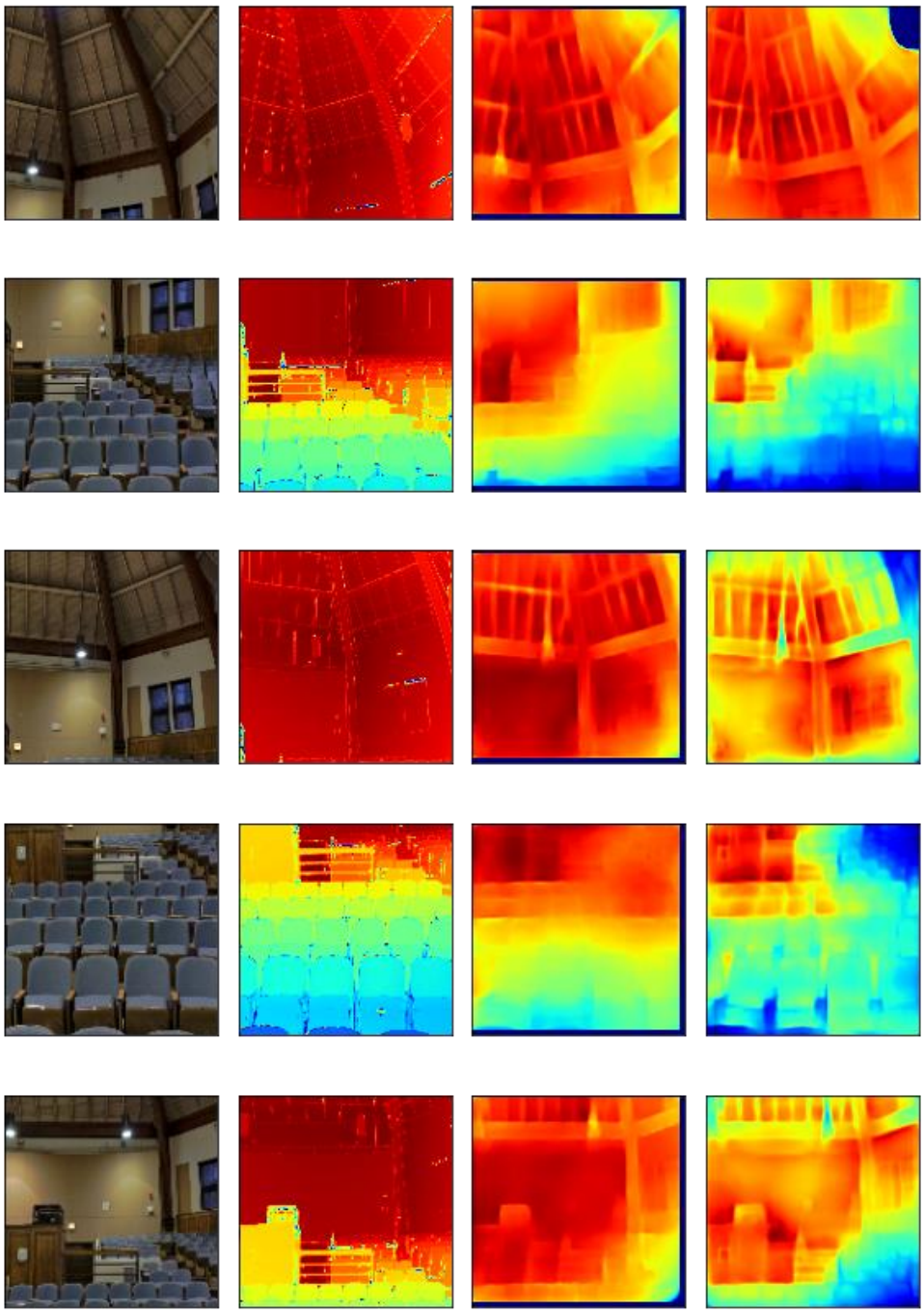
Im Folgenden sind die Ergebnisse des Szenarios dargestellt. Dazu ist jeweils ein Graph mit dem Train- und Validierungsloss zu sehen, sowie die Ausgaben des Modells, für drei Szenen, im Vergleich zum Erwartungswert.



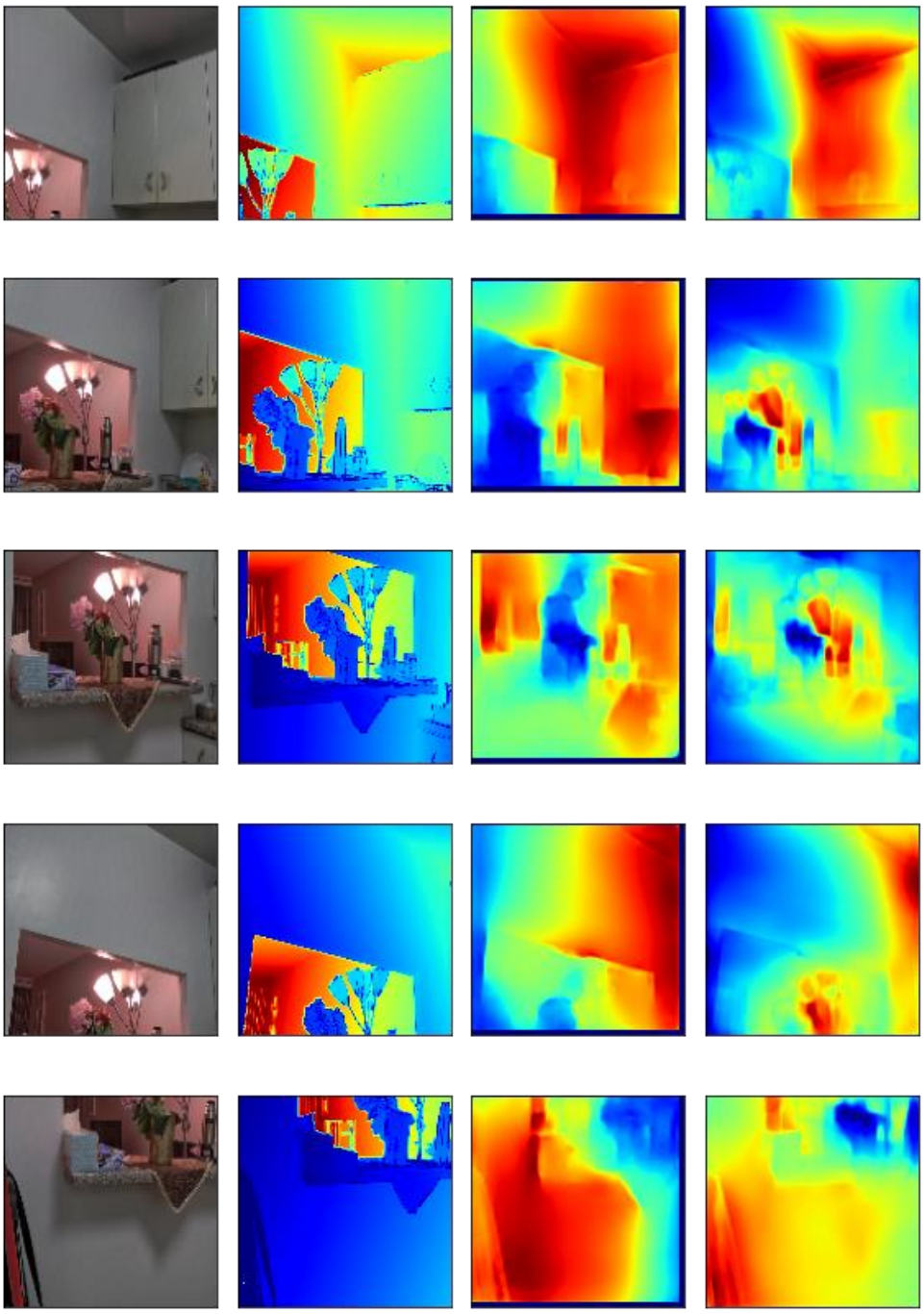
5-44 Train und Test Loss ohne Dropout



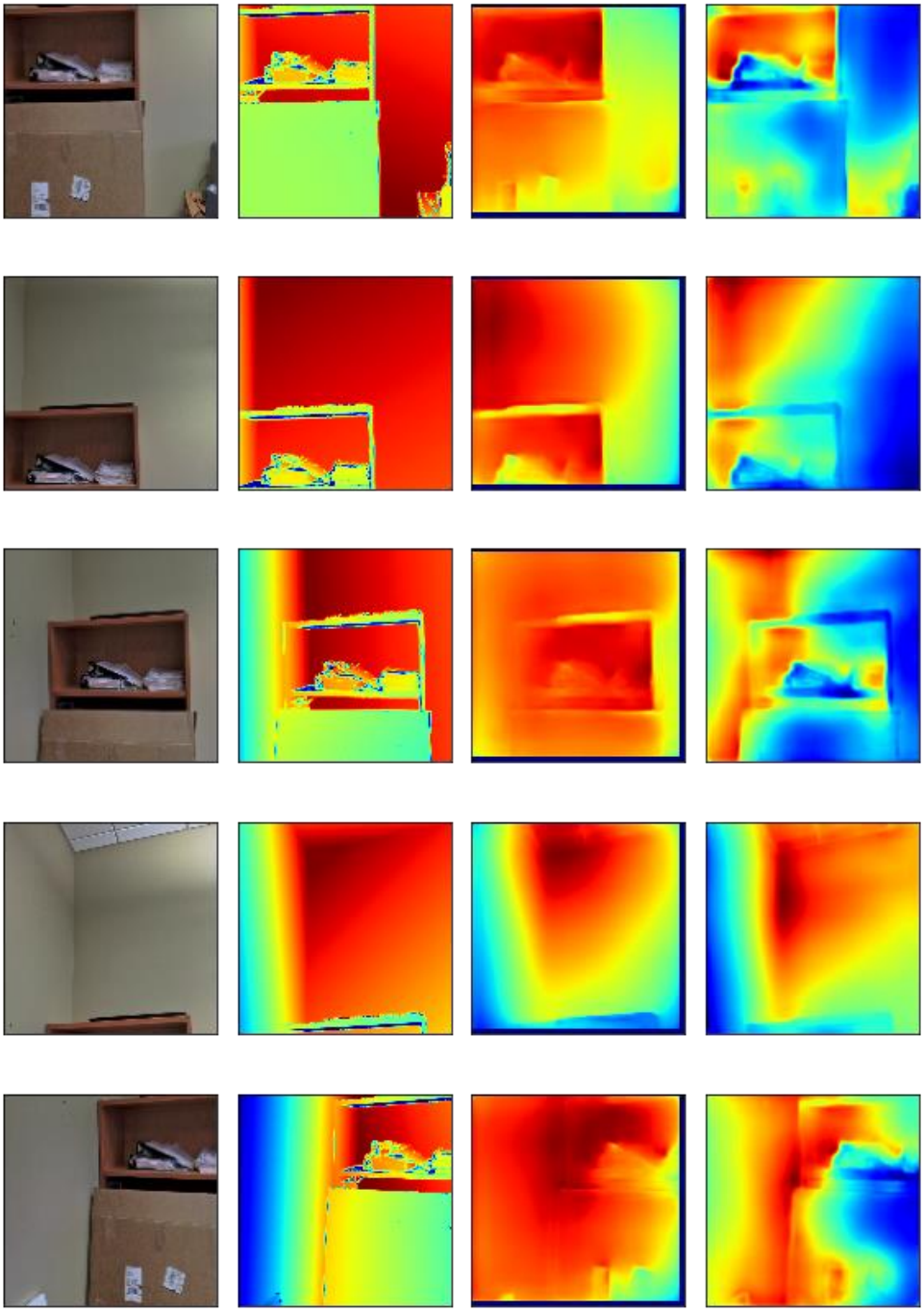
5-45 Train und Test Loss mit Dropout



5-46 Modellausgabe für Szene 1. Eingabebild(1.), Label(2.), Ohne Dropout(3.), Mit Dropout(4.)



5-47 Modellausgabe für Szene 2. Eingabebild(1.), Label(2.), Ohne Dropout(3.), Mit Dropout(4.)



5-48 Modellausgabe für Szene 3. Eingabebild(1.), Label(2.), Ohne Dropout(3.), Mit Dropout(4.)

Evaluation

Dieses Szenario zeigt wieder ein überraschendes Ergebnis. Die Loss-Metriken aus Abbildung *5-45* zeigen, dass der Train Loss zuerst über dem Test Loss liegt und anschließend sinkt. In Abbildung *5-44* ist der Train Loss jedoch durchgehend niedriger als der Test Loss. Aus der Theorie sollte also die Architektur mit den Dropout Layer bessere Ergebnisse aufweisen, wie

zuvor ist dies jedoch nicht der Fall. In Abbildungen *5-46* bis *5-48* ist klar zu sehen, dass die Dropout Layer ein schlechteres Ergebnis liefern. Es wurde vermutet, dass die Anordnung der Layer und die Position des Dropout Layers das Ergebnis verbessern könnten, was im nächsten Szenario untersucht wurde.

Szenario 5: Overfitting 3: Anordnung der Layer mit Dropout anpassen

Wie in Abschnitt *5.1.3* diskutiert, macht die Anordnung der Layer, insbesondere bei der Benutzung von Dropout, einen Unterschied. Dieser Unterschied wurde in diesem Szenario untersucht und die Ergebnisse werden vorgestellt. Es werden insgesamt drei Anordnungen miteinander verglichen. Eine davon aus Szenario 4.

Modellarchitektur

Die Modellarchitektur aus Szenario 4 wird übernommen und die Position der Layer wird verändert. Dabei gibt es drei Varianten:

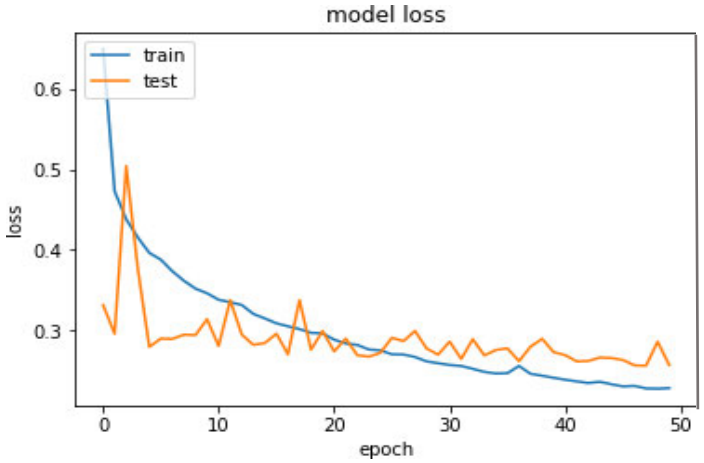
1. Conv2D → LeakyReLU → BatchNormalization → Dropout
2. Conv2D → BatchNormalization → LeakyReLU → Dropout
3. Conv2D → LeakyReLU → Dropout → BatchNormalization

Hyperparameter:

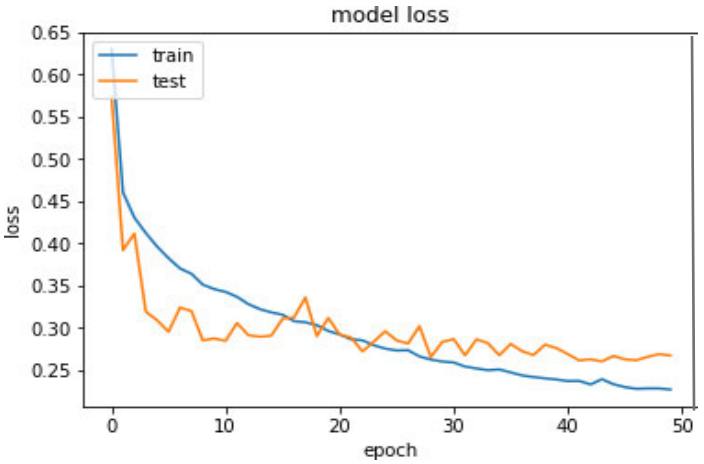
- Kernel Größe: 5x5
- Batch Size: 32
- Größe des Bottleneck: 256
- Epochen: 50
- Art des Bottlenecks: Convolutional
- Loss Funktion: Benutzerdefiniert
- Optimizer: Adam mit lr=0.0001
- Aktivierungsfunktion des letzten Layers: ReLU

Ergebnisse

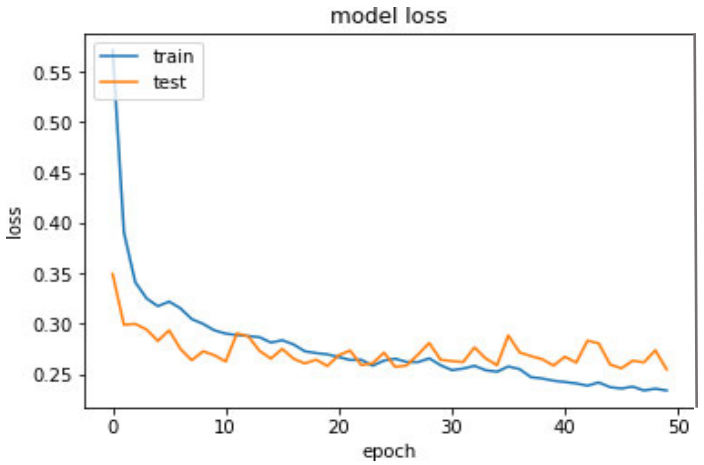
Die Ergebnisse sind in Form eines Graphen zu jeder Variante dargestellt, welcher den Train und Test Loss zeigt. Die Modellausgaben der drei Szenen werden, als Vergleich der Varianten, in den darauffolgenden Abbildungen gezeigt.



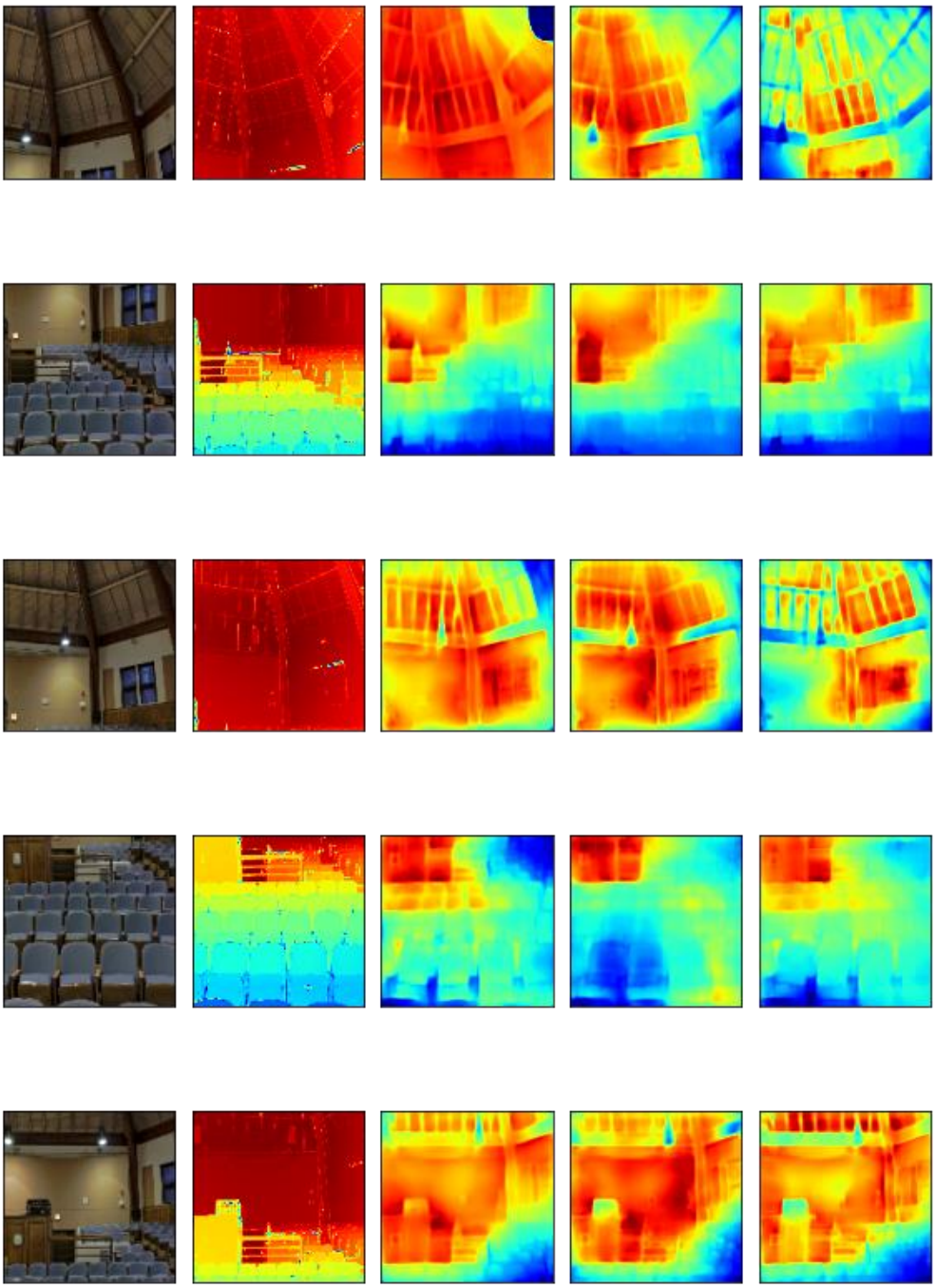
5-49 Train und Test Loss für Variante 1



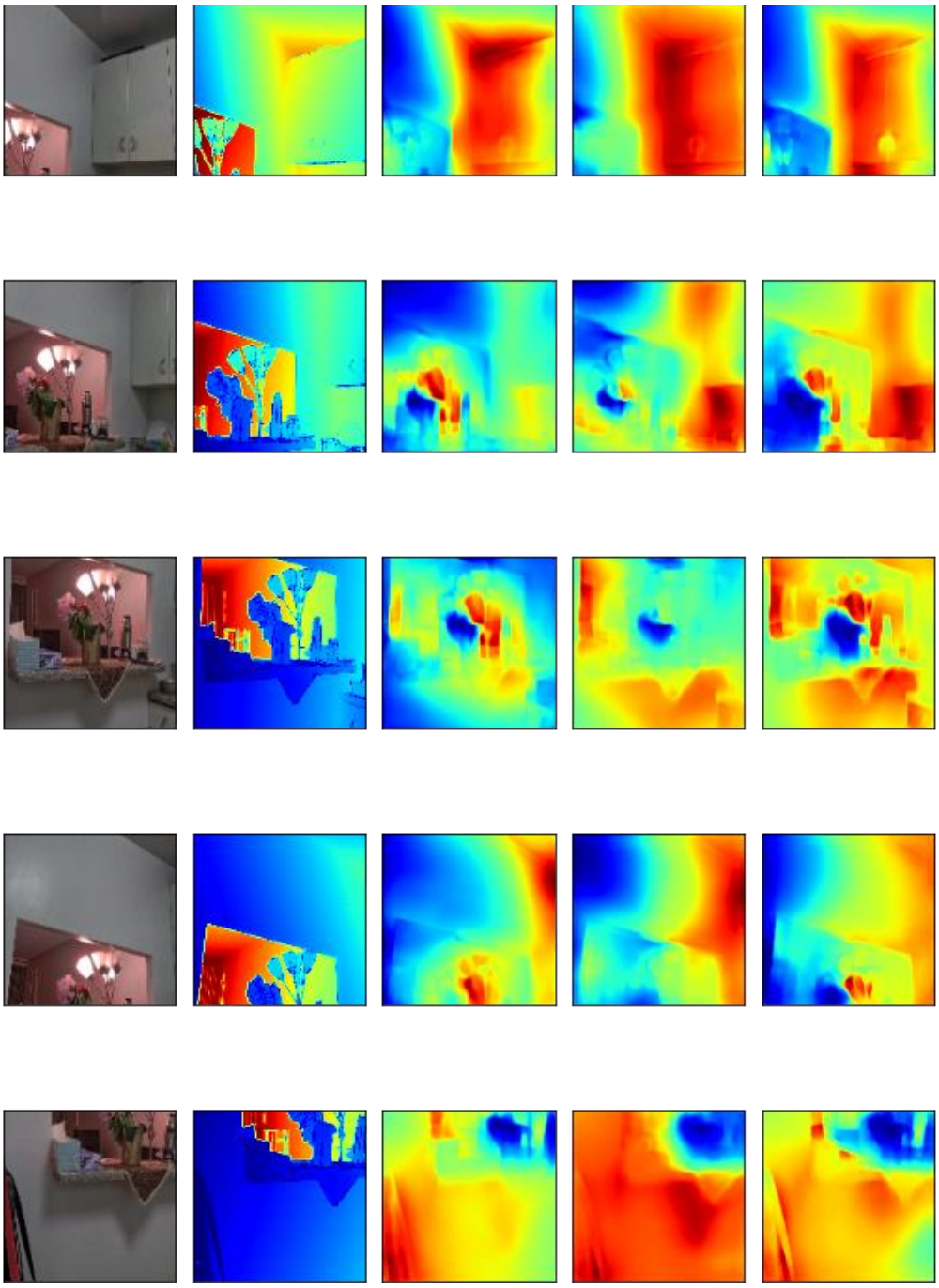
5-50 Train und Test Loss für Variante 2



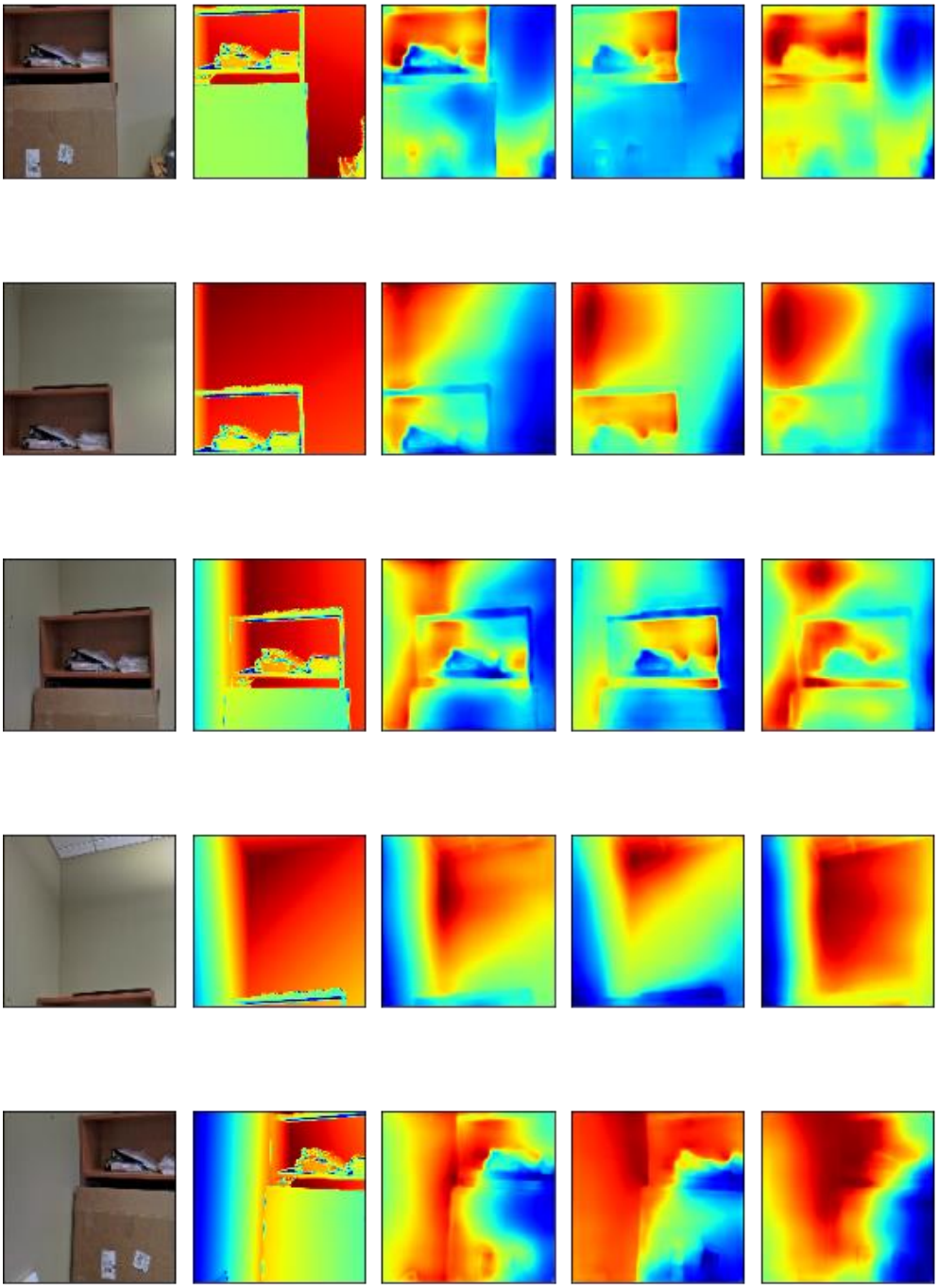
5-51 Train und Test Loss für Variante 3



5-52 Modellausgabe für Szene 1. Eingabebild(1.), Label(2.), Variante 1(3.), Variante 2(4.), Variante 3(5.)



5-53 Modellausgabe für Szene 2. Eingabebild(1.), Label(2.), Variante 1(3.), Variante 2(4.), Variante 3(5.)



5-54 Modellausgabe für Szene 3 Eingabebild(1.), Label(2.), Variante 1(3.), Variante 2(4.), Variante 3(5.)

Evaluation

Die Loss-Metriken (Abbildungen *5-49* bis *5-51*) und Modellausgaben (Abbildungen *5-52* bis *5-54*) zeigen leichte Unterschiede, jedoch keine Verbesserungen zu den vorherigen Szenarien. Rückblickend zur Diskussion aus Abschnitt *5.1.3*, macht es durchaus einen Unterschied, wie die Layer angeordnet sind, in diesem Fall allerdings nicht enorm. Möglicherweise stellen sich bei einem längeren Training oder anderen Anpassungen der Hyperparameter größere Veränderung heraus.

6 Zusammenfassung und Ausblick

In diesem Kapitel werden die Ergebnisse und Erkenntnisse dieser Arbeit zusammengefasst und anhand dessen mögliche weitere Vorgehen oder Entwicklungen der Tiefenbildrekonstruktion erläutert.

6.1 Fazit

Im Rahmen dieser Arbeit wurde die 3D-Tiefenbildrekonstruktion mit Hilfe von Faltungsnetzwerken, anhand des DIODE-Datensatzes, untersucht. Das Ziel der Tiefenbildrekonstruktion ist, aus einem RGB-Bild die Tiefe eines jeden Pixels vorherzusagen und somit eine Art Tiefenkarte, oder auch Tiefenbild, des Bildes zu erstellen. Bei der Konstruktion von Machine Learning Modellen, die mit Bildern arbeiten, haben sich CNN Modelle als der Standard etabliert. Bei der Tiefenbildrekonstruktion findet eine Rekonstruktion statt, sodass die Ausgabe des Modells nicht eine niedrigdimensionale Klassifizierung, sondern eine gleichdimensionale Ausgabe des Eingabebildes mit Tiefeninformationen ist. Aus diesen Gründen war auf die Modellarchitektur eines Convolutional Autoencoders nicht zu verzichten. Hineinblickend in das Endziel, ein Tiefenbild aus einem RGB-Bild zu erstellen, wurde ein Autoencoder entwickelt, um ein RGB-Bild zu rekonstruieren. Dies hatte zur Folge, dass sich mit der Entwicklung von Autoencodern vertraut gemacht und eine Basis für die Tiefenbildrekonstruktion geschaffen wurde. Der entwickelte Autoencoder für die Bild- und Tiefenbildrekonstruktion wurde anhand verschiedener Szenarien untersucht und bewertet.

Des Weiteren haben die Szenarien des Autoencoders für Bildrekonstruktion veranschaulicht, dass vermeintliche unbedeutende Hyperparameter einen großen Unterschied machen können und dessen Kombination zu beachten ist.

Die Szenarien des Autoencoders für Tiefenbildrekonstruktion haben gezeigt, dass die Aufgabe, ein Tiefenbild aus einem RGB-Bild vorherzusagen, eine schwere, aber nicht unmögliche ist. Übertragend aus der Vorarbeit mit der Bildrekonstruktion, wurde auch hier gezeigt, dass das Tuning der Hyperparameter essenziell ist. Die Änderung der Aktivierungsfunktion des letzten Layers und Vergrößern der Kernelgröße haben deutliche Besserungen aufgewiesen.

Das größte Problem scheint allerdings die Überanpassung zu sein. Mit Änderungen, wie die Verkleinerung des Bottlenecks und die Batch Size sowie die Nutzung von Dropout, wurde versucht gegen diese anzugehen. Rein von den Loss-Metriken her, schien dies teilweise zu funktionieren, diese aber oft getäuscht haben, da das entstandene Tiefenbild keine Besserung zeigte. Auch die Anordnung der Layer spielt in der Regel eine größere Rolle, dies ist anscheinend von Anwendungsfall zu Anwendungsfall unterschiedlich. Die Veränderung der Position der Aktivierungsfunktion- , BatchNormalization- und Dropout-Layer haben in diesem Fall keine großen Unterschiede gezeigt.

Zusätzlich wurde im Allgemein noch festgestellt, dass während einige rekonstruierte Tiefenbilder sehr vielversprechend erschienen, andere deutliche Ungenauigkeiten, im Vergleich zur Erwartung, zeigten. Relativ betrachtet wurden für die erste Szene, gegenüber der anderen beiden, konsistent die besten Ergebnisse festgestellt. Besonders die Tiefe der Stuhlreihe, im Hörsaal der ersten Szene, wurde oftmals richtig vorhergesagt. Zurückblickend auf das Shape from Texture Prinzip aus Abschnitt *2.1.2*, kann hier eine Verbindung gelegt werden. Da dieses Prinzip darauf beruht, dass die Verzerrung oder Verkleinerung einzelner Texturelemente, in einem 2D-Bild, auf die Veränderung dessen 3D-Form hindeutet, wird hier vermutet, dass die Texturen der Stuhlreihen ebenso darauf hindeuten. In anderen Bildern ist Ähnliches zu erkennen. In Szene 1 an der Decke des Hörsaals, sowie in Szene 2 und 3 and der Innenecke zwischen Wand und Decke, werden die Ecken als besonders Tief gesehen. In Szene 1 sind an der Decke wieder Texturmerkmale zu sehen, an denen erkannt werden kann, wie Tief einige Objekte in dem Bild sind. In den Szenen 2 und 3 jedoch sind eher weniger Merkmale vorhanden, die darauf hinweisen könnten. Hier kommt jedoch der Depth smoothnes Loss ins Spiel. Dieser nutzt, gewichtet durch die Gradienten des Labels, die Gradienten der Ausgabe, um den Loss zu berechnen. Zwar sind an den Wänden keine Texturmerkmale, doch der Gradient übermittelt, dass es an dieser Stelle tiefer wird. Gut zu sehen, war dies auch an einigen Kanten, wo der Gradient höher ist. Objekte, die im Vordergrund waren, konnten dadurch auch oftmals als Nah erkannt werden, wie in den Ergebnissen aus Szene 2 zu sehen ist.

Die Kombination aus Texturgradient und Tiefengradient haben zu einem Gradienten-dominierten Training geführt. Beispielbilder aus dem Draußen-Teil des Datensatzes zeigen ein gleichartiges Verhalten (siehe Anhang: *7*). Gebäude mit Texturmerkmalen, konnten richtig eingeschätzt werden, sowie Objekte im Vordergrund. Bäume im Vordergrund, waren schwierig zu differenzieren, da zwischen den Ästen immer wieder die Tiefe des hinter gelegenen Gebäudes hohe Gradienten-Werten aufwies. Daher sind Objekte ohne Lücken einfacher zu differenzieren.

Zusammenfassend kann festgehalten werden, dass das Modell in der Lage ist aus Tiefenbildern zu lernen und eine Vorhersage zu treffen sowie, dass die Ergebnisse zwar noch nicht besonders akkurat sind, aber durchaus einen großen Schritt in die richtige Richtung gehen. Jedoch bietet sich dieses Modell noch nicht für kritische Anwendungsfälle an, in denen die Tiefeneinschätzung präzise sein muss. Andere Anwendungsfälle wie kleine Roboter oder die 3D-Computervision im Allgemein, können jedoch daraus profitieren.

6.2 Ausblick

Das, aus dieser Arbeit entstandene, Modell bietet noch viele Verbesserungsmöglichkeiten, kann aber als Grundlage anderer Arbeiten im Bereich der 3D-Tiefenbildrekonstruktion genutzt werden. Aus den gewonnenen Erkenntnissen werden im folgenden Abschnitt kurz einige Verbesserungsmöglichkeiten oder Ideen zur Weiterverwendung vorgeschlagen.

Loss Funktion anpassen

Diese Arbeit hat gezeigt, dass der Loss einen enormen Einfluss auf das Training und die Ergebnisse hat. Eine Anpassung der Gewichtungen der drei kombinierten Loss Funktionen könnte dabei eine Besserung der Ergebnisse zeigen. Andernfalls könnte der Loss insgesamt verändert werden, indem die Kombinationen verschiedener Loss Funktionen untersucht wird.

Vortrainiertes Modell ändern

Das vortrainierte Modell zu ändern, sollte auch in Betracht gezogen werden. VGG16 ist zwar ein sehr beliebtes Modell, aber mit einem Erscheinungsjahr von 2014 auch ein relativ altes. Ein sehr tiefes Modell, wie DenseNet-169 erwies sich als besser (siehe Anhang: *7*), weshalb weitere Modelle, wie ResNet, Inception oder EfficientNet, welche von der Keras API angeboten werden, auch untersucht werden sollten.

Weitere Maßnahmen gegen Überanpassung

Da die Überanpassung das Hauptproblem bei den Ergebnissen war, sollte dies weiter untersucht werden. Neben der Nutzung von Dropout gibt es noch weitere Möglichkeiten dagegen anzugehen:

- Regularisierung
- Erhöhung der Datenmenge
- Früheres Stoppen des Trainings
- Modell verkleinern

Aufteilen der Aufgaben des Modells

Aus dieser Arbeit ist klar geworden, dass das Modell für eine gewisse Art von Bildern besonders gut funktioniert. Erweiterungen der Arbeit könnten die Aufteilung der Ziele des Modells sein, indem größere Datensätze für konkrete Szenen genutzt werden. Neben dem Draußen-Teil, den dieser Datensatz schon mit sich bringt, könnte ein Datensatz genutzt werden, welcher hauptsächlich Szenen mit vielen Texturen, die eine Veränderung aufweisen, beinhält oder einen mit weniger Texturen. Für jedes dieser Datensätze könnten Modellansätze, aufbauend auf diesem, untersucht werden, sodass darauffolgende Arbeiten ein akkurates Modell zur Tiefenbildrekonstruktion jeglicher Szenen untersuchen und erstellen können. Zusätzlich bietet dieser Datensatz auch Bilder mit Normalen an, auf dem eine Arbeit beruhen könnte.

7 Anhang

In diesem Kapitel werden einige Modellausgaben gezeigt, die eine Relevanz zur Arbeit, besonders zu Kapitel *6,* haben. Diese waren allerdings nicht Teil der Szenarien oder Ausarbeitung, sind jedoch trotzdem interessant, weshalb sie in dem Anhang gezeigt werden.

DIODE-Outdoor

Nachfolgend werden Loss Metriken und Modellausgaben gezeigt, die den Außerhalb-Teil des Datensatzes verwendet haben. Die Modellarchitektur aus Szenario 3 wurde genutzt. Dazu wurden drei Szenen ausgewählt:

- Szene 1: Ein Campus o.ä., auf dem ein Gebäude sowie Stufen zu sehen sind.
- Szene 2: Ein Gehweg mit vielen Bäumen und einer Laterne im Vordergrund sowie Gebäuden im Hintergrund.
- Szene 3: Wohnungen mit einem Balkongerüst und einigen Bäumen

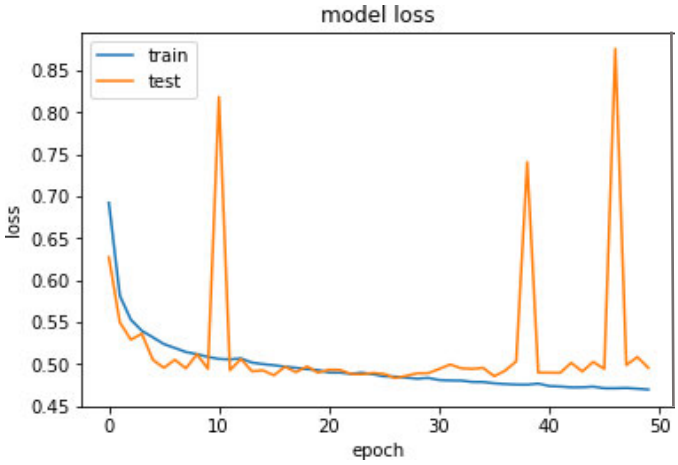


Abbildung A.1 Train und Test Loss für Outdoor-Daten

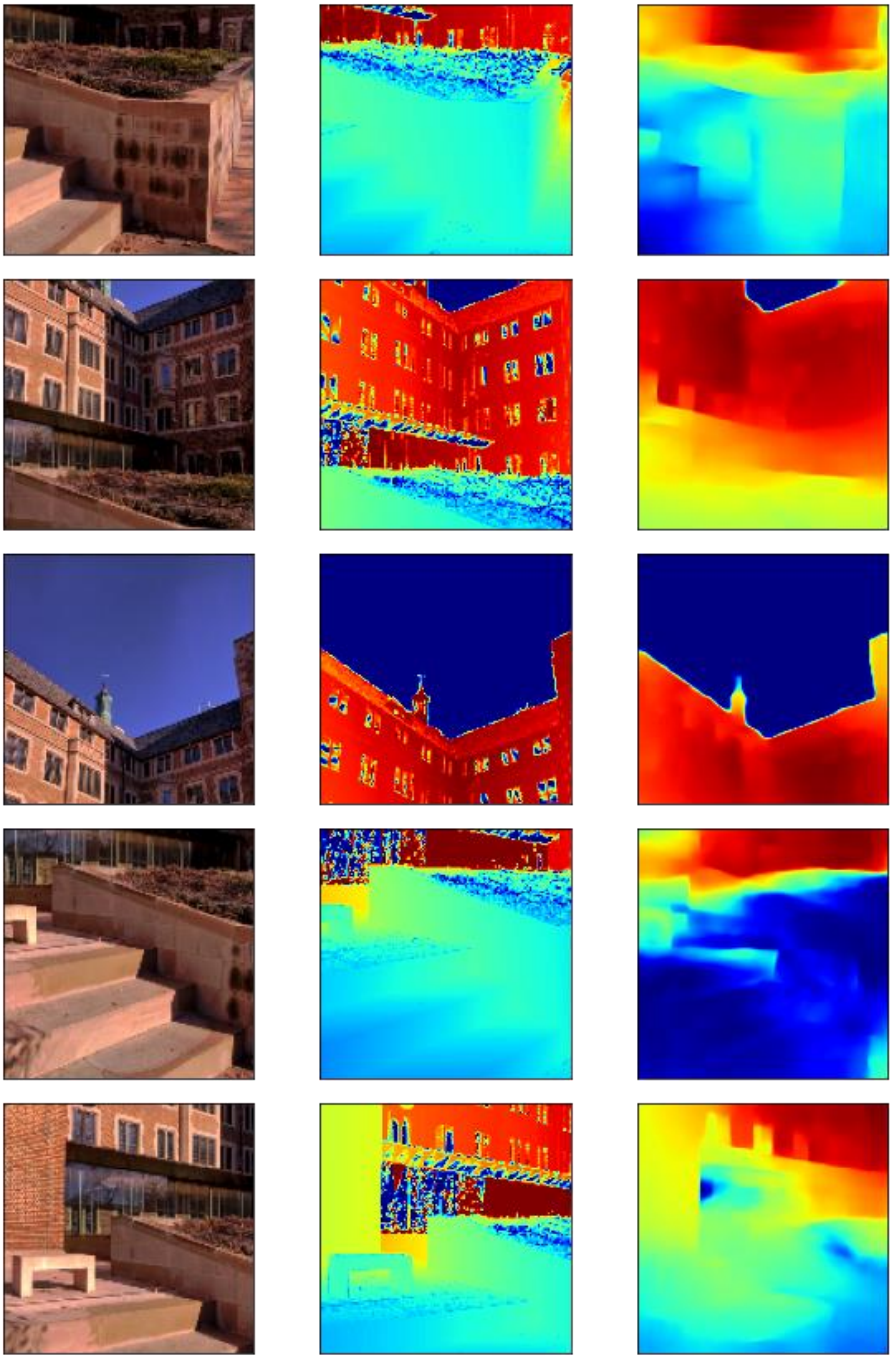


Abbildung A.2 Modellausgaben für Szene 1. Eingabebild(1.), Label(2.), Ausgabe(3.)

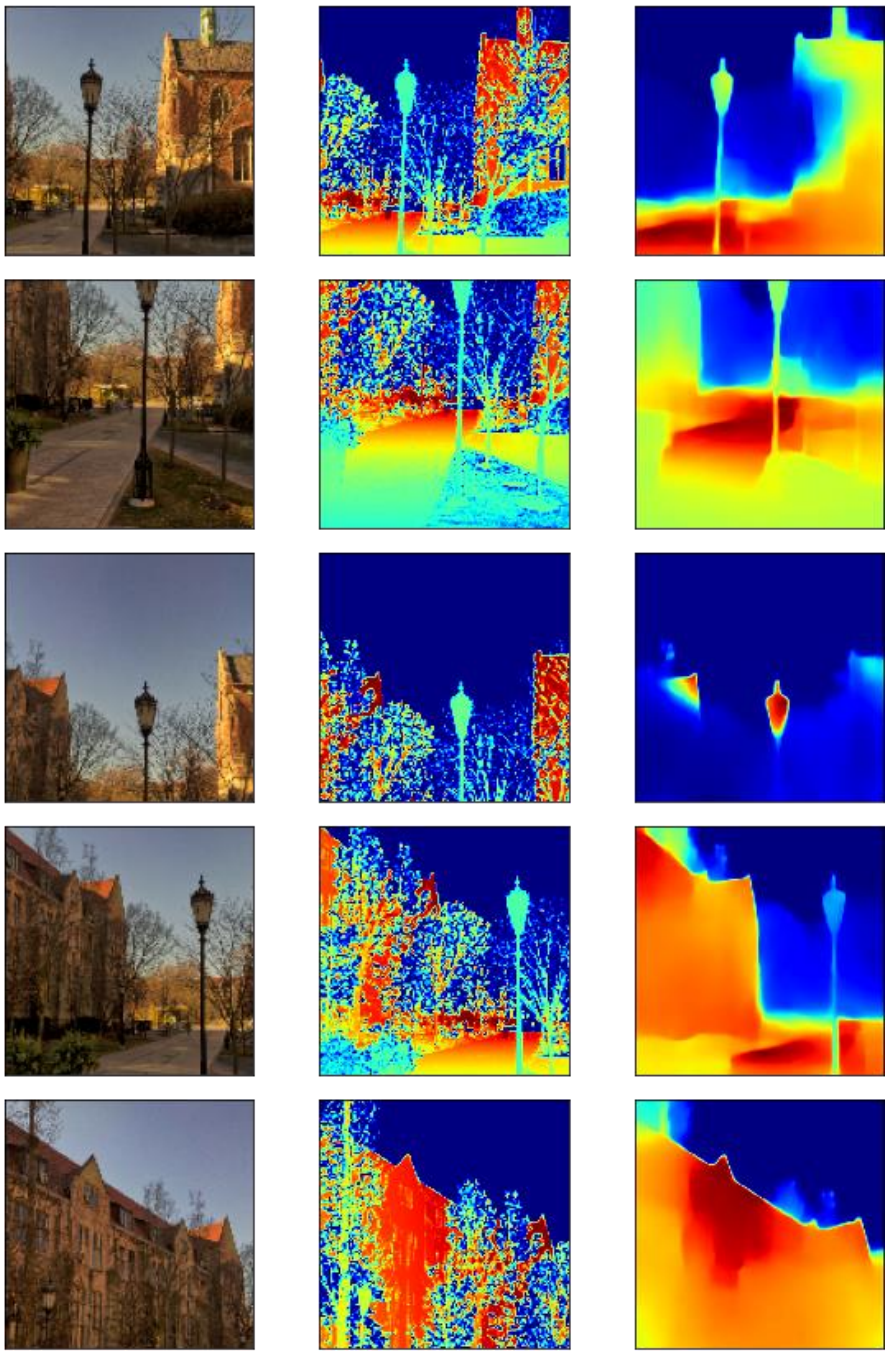


Abbildung A.3 Modellausgaben für Szene 1. Eingabebild(1.), Label(2.), Ausgabe(3.)

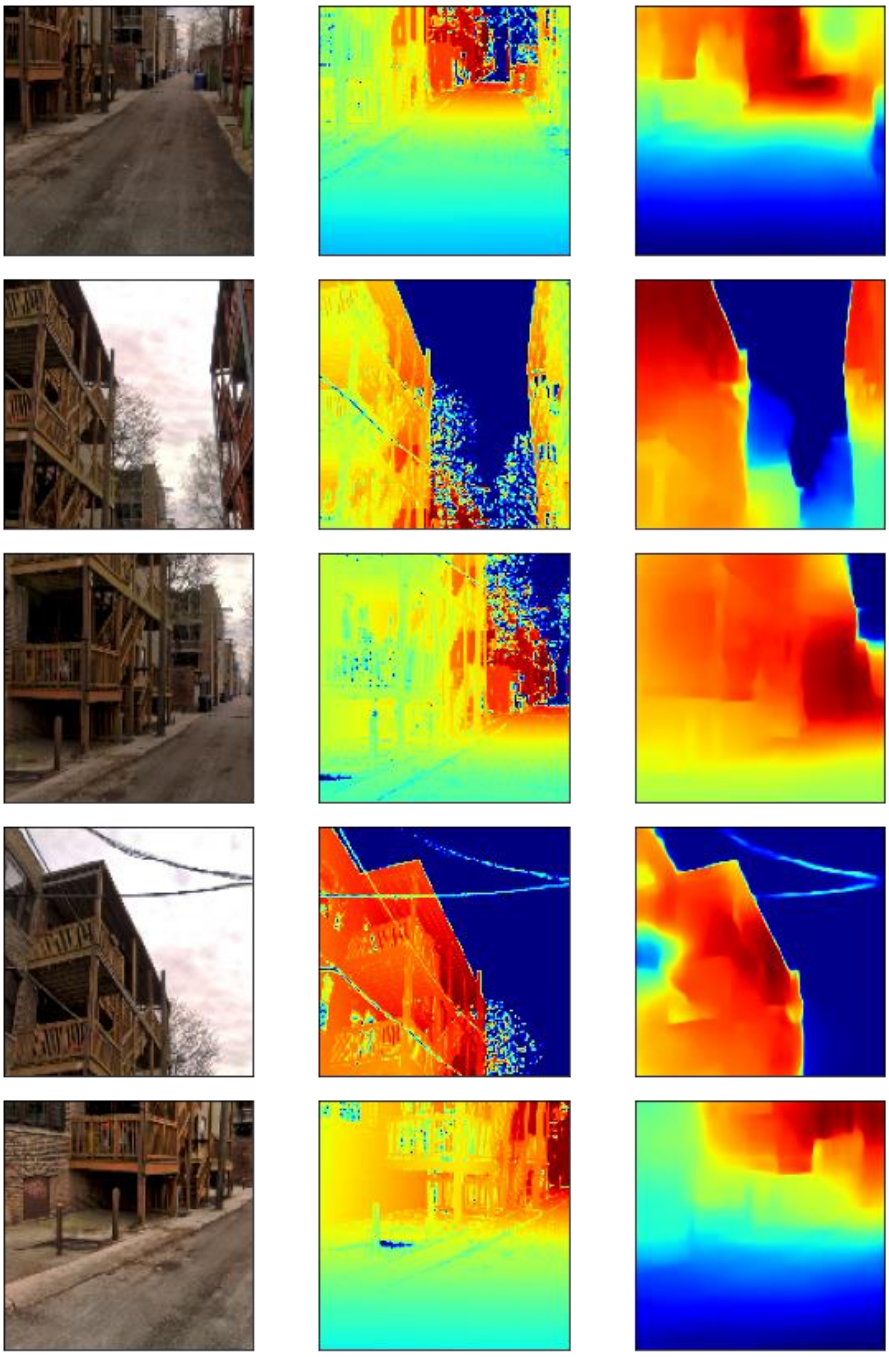


Abbildung A.4 Modellausgaben für Szene 1. Eingabebild(1.), Label(2.), Ausgabe(3.)

DenseNet-169

In diesem Abschnitt werden die Loss Metriken und Modellausgaben für das Transfer Learning mit DenseNet-169 gezeigt. Dabei wurde der Modellaufbau von (Alhashim, et al., 2019) übernommen und ansonsten die Hyperparameter aus Szenario 3.

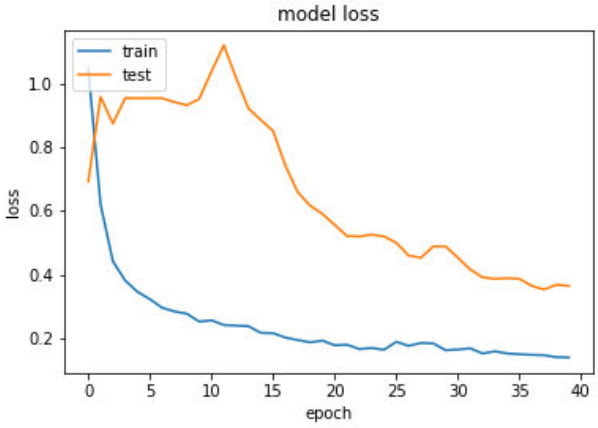


Abbildung A.5 Train und Test Loss für Outdoor-Daten

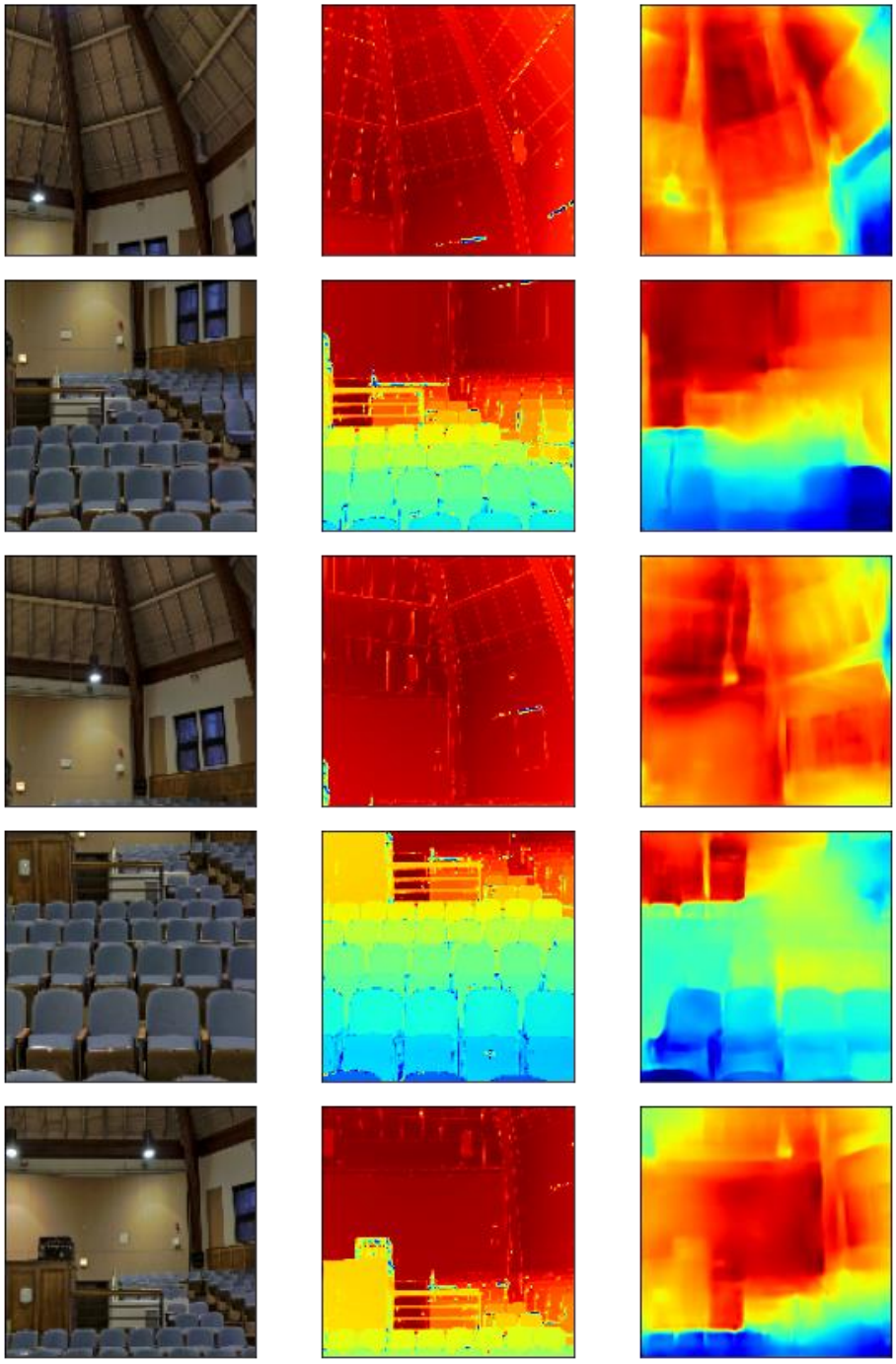


Abbildung A.6 Modellausgaben für Szene 1. Eingabebild(1.), Label(2.), Ausgabe(3.)

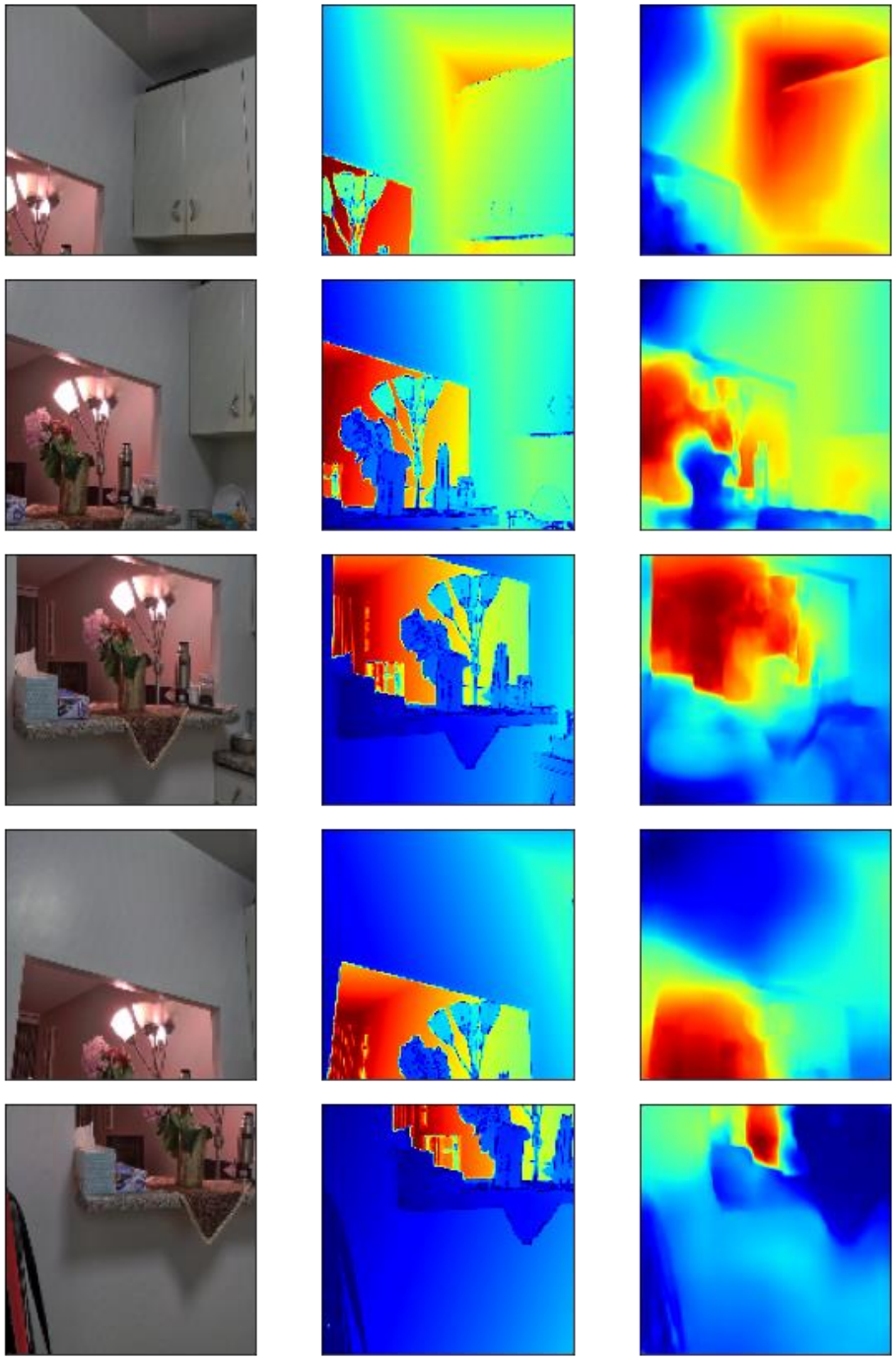


Abbildung A.7 Modellausgaben für Szene 1. Eingabebild(1.), Label(2.), Ausgabe(3.)

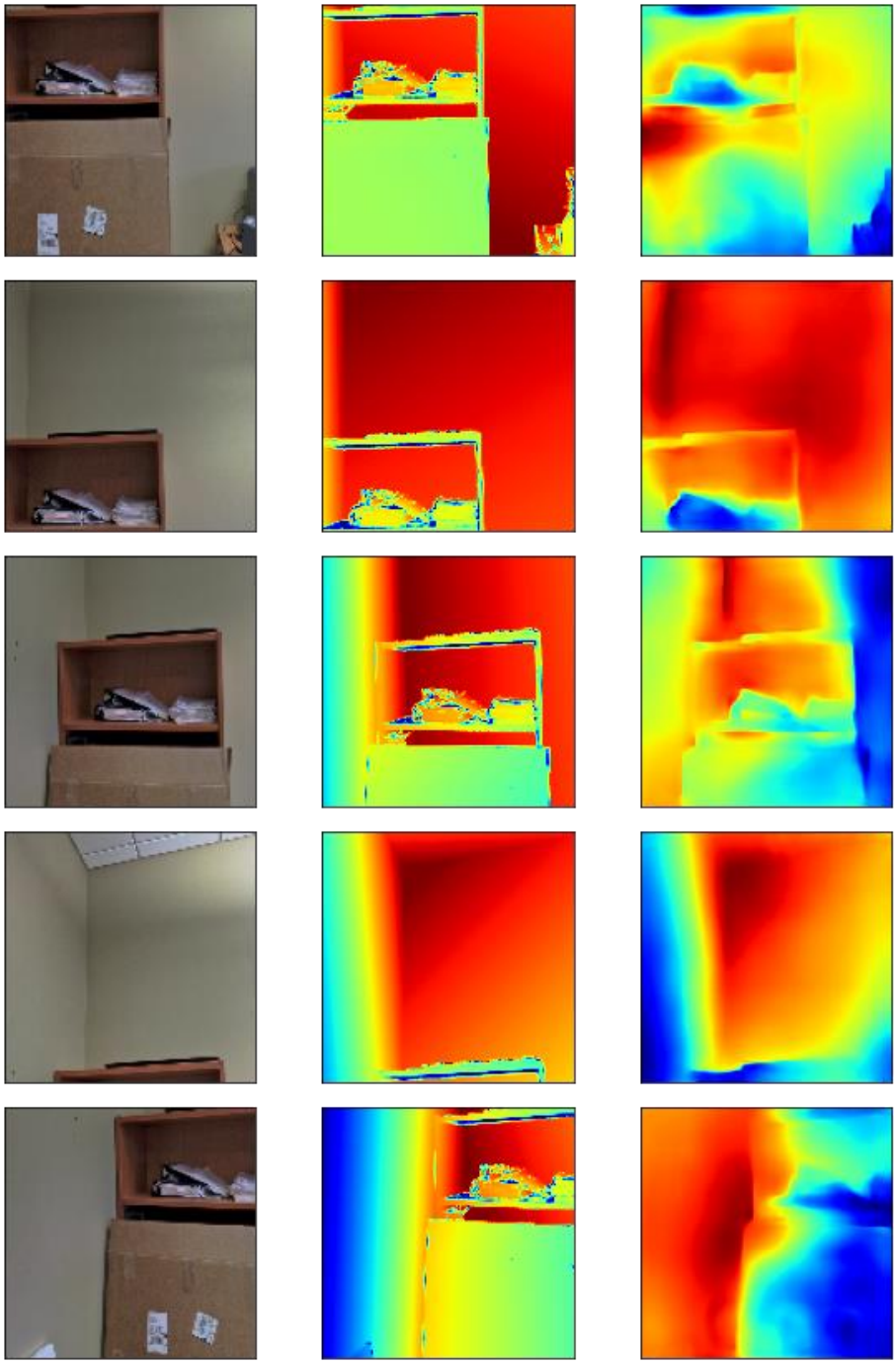


Abbildung A.8 Modellausgaben für Szene 1. Eingabebild(1.), Label(2.), Ausgabe(3.)

DIODE-Normals

Im Folgenden werden die Loss Metriken und Modellausgaben für den Datensatz der Normalen dargestellt. Dafür wurden die der Szene aus den Szenarien in *5.2.4* gewählt.

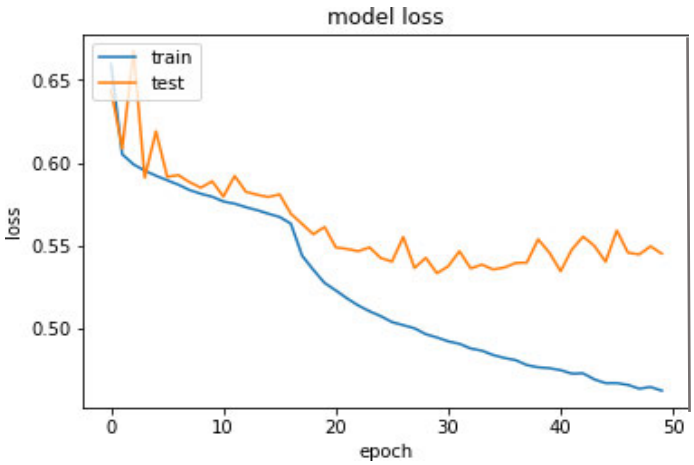


Abbildung A.9 Train und Test Loss für Normalen

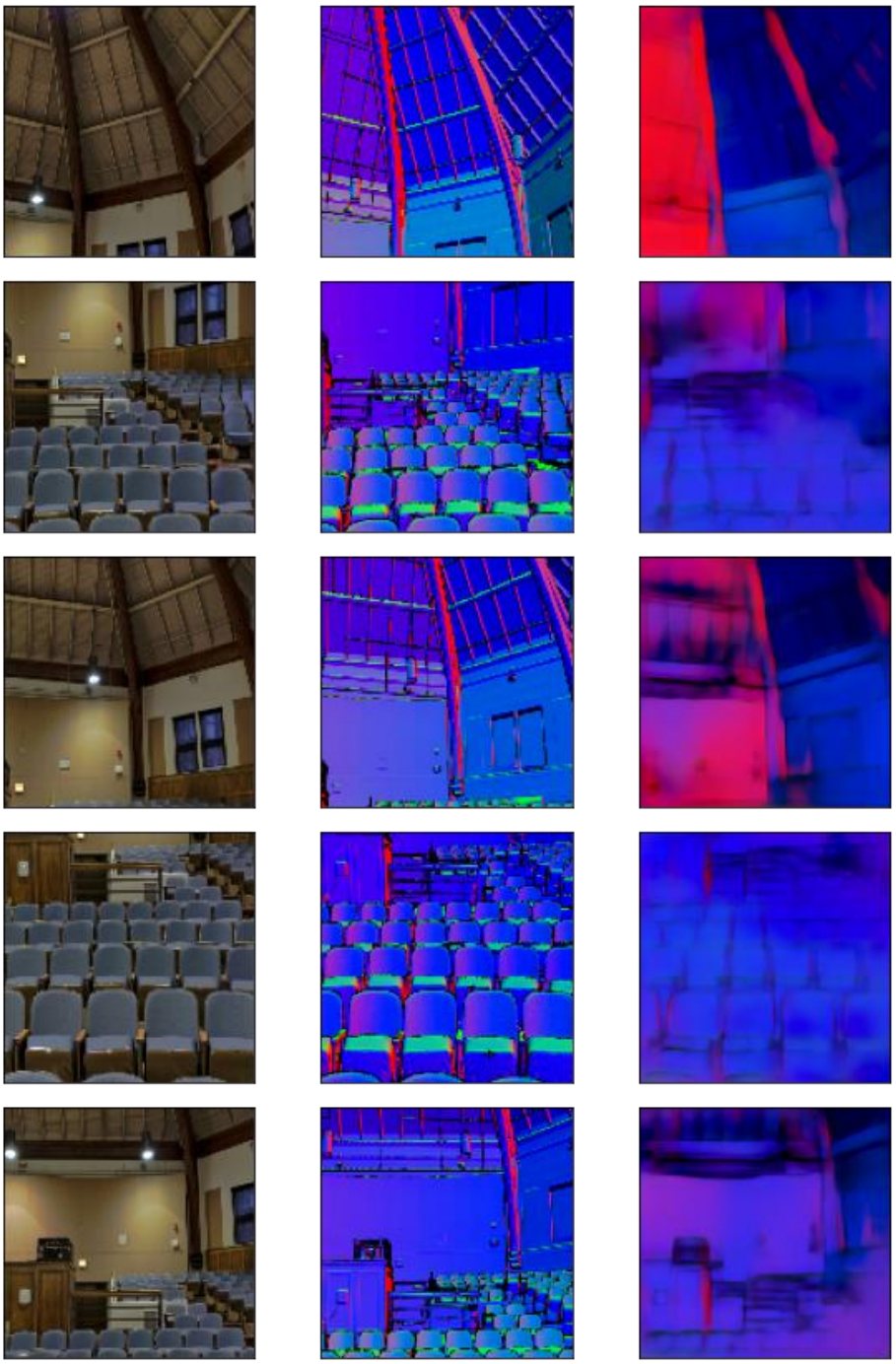


Abbildung A.10 Modellausgaben für Szene 1. Eingabebild(1.), Label(2.), Ausgabe(3.)

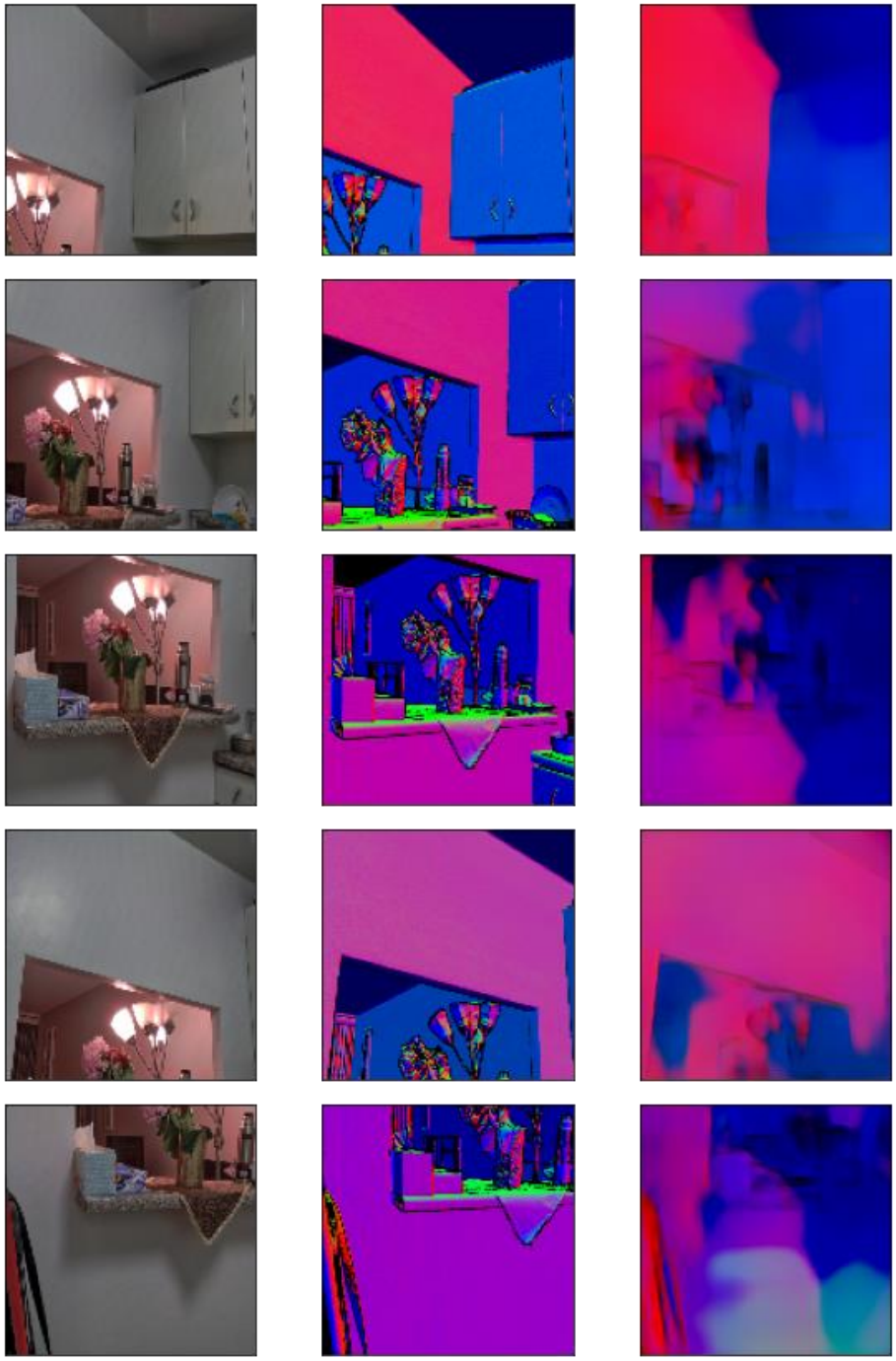


Abbildung A.11 Modellausgaben für Szene 1. Eingabebild(1.), Label(2.), Ausgabe(3.)

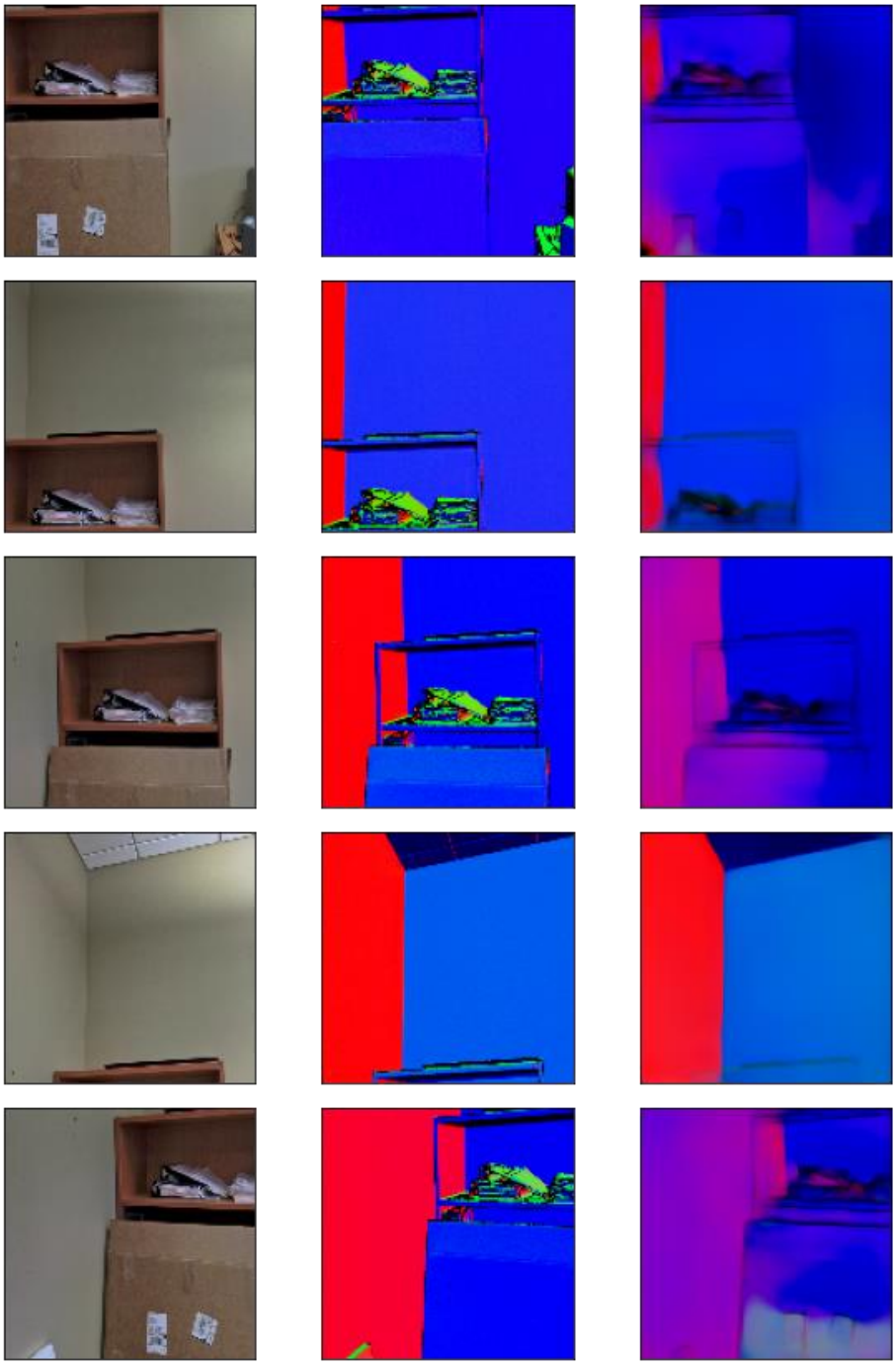


Abbildung A.12 Modellausgaben für Szene 1. Eingabebild(1.), Label(2.), Ausgabe(3.)

Literaturverzeichnis

Adobe. Erkärung des Begriffs Vektorgrafik. - URL https://www.adobe.com/de/creativecloud/design/discover/vector-file.html#:~:text=Die%20g%C3%A4ngigsten%20Vektordateitypen%20sind%3A,PostScript%20ist%20ein%20%C3%A4lteres%20Vektorgrafikformat. - Zugriff: 2022.

Alhashim I. und Wonka P. High Quality Monocular Depth Estimation via Transfer Learning. - URL https://arxiv.org/pdf/1812.11941.pdf. - 2019.

Anonym 7 beliebte Aktivierungsfunktionen, die Sie in Deep Learning kennen sollten und wie Sie sie mit Keras und TensorFlow 2 verwenden können. - URL https://ichi.pro/de/7-beliebte-aktivierungsfunktionen-die-sie-in-deep-learning-kennen-sollten-und-wie-sie-sie-mit. - Zugriff: 2022.

Ardabili S. [et al.] Deep Learning and Machine Learning in Hydrological Processes, Climate Change and Earth Systems: A Systematic Review. - URL https://www.researchgate.net/publication/335206371. - 2019.

Basu V. Monocular depth estimation. - URL https://keras.io/examples/vision/depth_estimation/. - 2021.

Bruker. Fokus-Variation Das technische Prinizp. - URL https://www.alicona.com/de/unsere-technologie/fokus-variation/. - Zugriff: 2022.

Casser V. [et al.] Depth Prediction Without the Sensors: Leveraging Structure for Unsupervised Learning from Monocular Videos. - URL https://arxiv.org/pdf/1811.06152v1.pdf. - 2018.

Chen G. [et al.] Rethinking the Usage of Batch Normalization and Dropout in the Training of Deep Neural Networks. - URL https://arxiv.org/pdf/1905.05928.pdf. - 2019.

Chollet F GitHub. - URL https://github.com/keras-team/keras/issues/1802#issuecomment-187966878. - 2016.

Chollet F. Building Autoencoders in Keras. - URL https://blog.keras.io/building-autoencoders-in-keras.html. - 2016.

Creswell A., Arulkumaran K. und Bharrath A. On denoising autoencoders trained to minimise binary cross-entropy. - URL https://arxiv.org/pdf/1708.08487.pdf. - 2017.

Dertat A. Applied Deep Learning - Part 3: Autoencoders. - URL https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798. - 2017.

Djib2011 Stackoverflow. - URL https://datascience.stackexchange.com/questions/38118/what-is-the-difference-between-upsampling-and-bi-linear-upsampling-in-a-cnn. - 2018.

Eigen D. und Fergus R. Predicting Depth, Surface Normals and Semantic Labels with a Common Multi-Scale Convolutional Architecture. - URL https://arxiv.org/pdf/1411.4734v4.pdf. - 2015.

Eigen D., Puhrsch C. und Fergus R. Depth Map Prediction from a Single Image using a Multi-Scale Deep Network. - URL https://arxiv.org/pdf/1406.2283.pdf. - 2014.

erentar. Jupyter kernels. - URL https://github.com/jupyter/jupyter/wiki/Jupyter-kernels. - 2022.

Geiger A. [et al.] Depth Evaluation. - URL http://www.cvlibs.net/datasets/kitti/eval_depth_all.php. - Zugriff: 2022.

Gholamy A., Kreinovich V. und Kosheleva O. Why 70/30 ord 80/20 Relation Between Training and testing Sets: A Pedagogical Explanation. - URL https://scholarworks.utep.edu/cgi/viewcontent.cgi?article=2202&context=cs_techrep. - 2018.

Google Google Scholar Lookup. - URL https://scholar.google.com/scholar_lookup?arxiv_id=1411.4734. - Zugriff: 2022.

HAW Hamburg. JuptyerHub. - URL https://jupyterhub.informatik.haw-hamburg.de/. - Zugriff: 2020.

He K. [et al.] Deep Residual Learning for Image Recognition. - URL https://arxiv.org/pdf/1512.03385.pdf. - 2015.

Huang G. [et al.] Densely Connected Convolutional Networks. - URL https://arxiv.org/pdf/1608.06993.pdf. - 2018.
Ioffe S. und C. Szegedy Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. - URL https://arxiv.org/pdf/1502.03167.pdf. - 2015.
Jing Y und Guanci Y Modified Convolutional Neural Network Based on Dropout and the Stochastic Gradient Descent Optimizer. - URL https://www.researchgate.net/publication/323617663. - 2018.
Keras Why choose Keras? - URL https://keras.io/why_keras/. - Zugriff: 2022.
Keras. Keras. - URL https://keras.io/. - 2015.
Keskar N. [et al.] ON LARGE-BATCH TRAINING FOR DEEP LEARNING: GENERALIZATION GAP AND SHARP MINIMA. - URL https://arxiv.org/pdf/1609.04836.pdf. - 2017.
Kingma D. und Ba J. ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION. - URL https://arxiv.org/pdf/1412.6980.pdf. - 2017.
Krantz J. Texture Gradient. - URL https://psych.hanover.edu/krantz/art/texture.html. - Zugriff: 2022.
Li X. [et al.] Understanding the Disharmony between Dropout and Batch Normalization by Variance Shift. - URL https://arxiv.org/pdf/1801.05134.pdf. - 2018.
Linux User PAKETE VERWALTEN. - URL https://www.linux-community.de/ausgaben/linuxuser/2018/02/gut-vernetzt/3/. - 2018.
Meisel A. HAW Hamburg, Mustererkennung und Machine Learning: Kursmaterialien. - 2021.
Microsoft Xbox Kinect. - URL https://www.xbox.com/de-DE/kinect. - Zugriff: 2020.
Papers with Code Datasets. - URL https://paperswithcode.com/datasets?task=monocular-depth-estimation. - Zugriff: 2022.
Project Jupyter. - URL https://jupyter.org/. - Zugriff: 2020.
Python. Python 3.9.7. - URL https://www.python.org/downloads/release/python-397/. - 2021.
Ronaghan S. Deep Learning: Which Loss and Activation Functions should I use? - URL https://towardsdatascience.com/deep-learning-which-loss-and-activation-functions-should-i-use-ac02f1c56aa8. - 2018.
Rosenblatt F. THE PERCEPTRON: A PROBABILISTIC MODEL FOR INFORMATION STORAGE AND ORGANIZATION IN THE BRAIN. - URL https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.335.3398&rep=rep1&type=pdf. - 1958.
Rosset S., Zhu J. und Hastie T. Margin Maximizing Loss Functions. - URL https://proceedings.neurips.cc/paper/2003/file/0fe473396242072e84af286632d3f0ff-Paper.pdf. - 2003.
Silberman N. [et al.] NYU Depth Dataset V2. - URL https://cs.nyu.edu/~silberman/datasets/nyu_depth_v2.html. - 2012.
Škrabánek P. und Zahradnikova jr., A. Automatic assessment of the cardiomyocyte development stages from confocal microscopy images using deep convolutional networks. - URL https://www.researchgate.net/publication/333502622. - 2019.
Solomonoff R. AN INDUCTIVE INFERENCE MACHINE. - URL http://www.raysolomonoff.com/publications/An%20Inductive%20Inference%20Machine1957.pdf. - 1957.
Sourtzinos P. Shape from Texture. - URL https://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/AV0506/s0565925.pdf. - Zugriff: 2022.
Srivastava N. [et al.] Dropout: A Simple Way to Prevent Neural Networks from Overfitting. - URL https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf. - 2014.
Ummenhofer B. [et al.] DeMoN: Depth and Motion Network for Learning Monocular Stereo. - URL https://arxiv.org/pdf/1612.02401.pdf. - 2017.
Vasilijevic I. [et al.] DIODE: A Dense Indoor and Outdoor DEpth Dataset. - URL https://arxiv.org/pdf/1908.00463.pdf. - 2019.

Vasilijevic I. [et al.] DIODE: A Dense Indoor and Outdoor DEpth Dataset. - URL https://diode-dataset.org/. - 2019.
Wang Z. [et al.] Image Quality Assessment: From Error Visibility to Structural Similarity. - URL https://www.cns.nyu.edu/pub/eero/wang03-reprint.pdf. - 2004.
Weiss K., Khoshgoftaar T. und Wang D. A survey of transfer learning. - URL https://journalofbigdata.springeropen.com/track/pdf/10.1186/s40537-016-0043-6.pdf. - 2016.
Wuttke L Künstliche Neuronale Netzwerke: Definition, Einführung, Arten und Funktion. - URL https://datasolut.com/neuronale-netzwerke-einfuehrung/. - Zugriff: 2022.
Zhuang F. [et al.] A Comprehensive Survey on Transfer Learning. - URL https://arxiv.org/pdf/1911.02685.pdf. - 2019.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Hamburg	30. März 2022	[redacted]
Ort	Datum	Unterschrift im Original