RGB-Sensormodelle anhand virtueller Umgebungen erlernen

Bachelorarbeit

Jan Christian Mackenstein

Prüfer der Bachelorarbeit:	1. Prof. Michael Beetz PhD
	2. Dr.-Ing. Dipl.-Inform. Thomas Röfer
Betreuer:	Andrei Haidu

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Bremen, den 18. Juli 2019

Jan Christian Mackenstein

Kurzfassung

Deep Learning bietet ganz neue Perspektiven in Sachen Flexibilität, Produktivität und Kosteneffizienz. In der vorliegenden Arbeit wird es genutzt, RBG-Sensormodelle mithilfe virtueller Umgebungen zu erlernen, damit in Zukunft für das Training von auf RBG-Sensordaten angewiesenen Robotern keine realen Umgebungen mehr notwendig sind. Die computergenerierten Bilder müssen hohen Qualitätsansprüchen genügen. Um dieses Ziel zu erreichen, werden in dieser Arbeit Generative Adversariale Netze (GAN) verwendet. Es werden verschiedene Möglichkeiten betrachtet, CycleGAN-Modelle mithilfe virtueller Umgebungen aus Unreal Engine zu trainieren, um möglichst realitätsnahe Abbildungen der virtuellen Umgebungen zu generieren. Für aufgetretene Probleme wurde durch Änderung von Parametern nach Lösungen gesucht.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Der Mensch nimmt verschiedene Reize aus der Umwelt über seine Sinne auf. Die Rezeptoren der Sinnesorgane wandeln diese Reize in elektrische Impulse um und leiten sie an das Gehirn weiter, wo sie verarbeitet werden. Das Ergebnis der Informationsgewinnung und - verarbeitung ist die Wahrnehmung, auch Perzeption genannt, und erfolgt beim Menschen primär visuell. Dabei spielen nicht nur visuelle Eigenschaften wie Form, Farbe, Oberfläche und Größe eine Rolle, sondern auch die Lokalisation des wahrgenommenen Objektes im Raum.

Zum besseren Verständnis nun einige physikalische und biologische Hintergründe. Licht ist die Basis dafür, dass ein Objekt abgebildet werden kann, dazu muss es ins Auge oder in eine Kamera gelangen. Trifft Licht auf ein Objekt werden bestimmte Lichtfrequenzen absorbiert, andere reflektiert. Damit der Mensch ein Objekt sieht, muss Licht auf die Netzhaut des Auges gelangen, wo sich zwei Arten von Fotorezeptoren befinden, Stäbchen für das Hell-Dunkel-Sehen und Zapfen für das Sehen von Farben. Es gibt drei Zapfentypen für die Lichtkomponenten Rot, Grün und Blau (RGB). Die Fotorezeptoren leiten über den Sehnerv Signale an das Gehirn, wo diese in Sekundenbruchteilen verarbeitet und durch das Einfließen von individuellen Erfahrungen zu einen Bild zusammengefügt werden. Durch die Kombination der Bildeindrücke beider Augen entsteht eine räumliche Wahrnehmung der Umgebung.

In der Funktionsweise ist eine Kamera dem menschlichen Auge ähnlich. Durch das Kameraobjektiv wird das Licht, ähnlich wie auf dem Weg zur Netzhaut, mehrfach gebrochen, um ein scharfes Abbild des Objektes auf dem Sensor zu erreichen. Bei einem Bildsensor entsprechen die optoelektronischen Bilderfassungselemente den Fotorezeptoren im Auge und wandeln wie diese das Licht in elektrische Signale um. Im Gegensatz zum Auge können einige Bildsensoren das gesamte Lichtspektrum erfassen.

Das leistungsfähige menschliche Sehsystem ist gut im Erkennen von Objekten sowie bei der 3D-Rekonstruktion, selbst dann wenn Teile fehlen. Maschinen hingegen eignen sich für präzise Messungen der visuellen Eigenschaften von Objekten. Durch ihre Objektivität sind sie gut geeignet zur Erkennung von Farben, Farbintensitäten und Grauwerten. Auch Reproduzierbarkeit ist eine Stärke von Maschinen.

Um die oben genannten Vorteile zu nutzen, werden in der vorliegenden Arbeit Bilder mit einer Kinect-Kamera aufgenommen, die über einen Tiefen- und einen RGB-Sensor verfügt.

RGB-Sensormodelle sollen hier genutzt werden, um anhand virtueller Umgebungen trainiert zu werden. So wie der Mensch im Gehirn Seherfahrungen mit Gelerntem verknüpft, soll auch die Künstliche Intelligenz Modelle erlernen, um neue Szenen und Objekte verarbeiten zu können.

Eine einzelne Nervenzelle wäre nicht intelligent, doch beim Menschen zählt das Gehirn viele Milliarden Neurone, die ständig neu verknüpft werden. Deep-Learning-Verfahren orientieren sich grob an dieser Arbeitsweise. Wie das Gehirn bestehen sie aus einem Netzwerk interagierender „Neurone“, die mehrere hierarchisch angeordnete Schichten bilden. Schichtweise werden „Neurone“ beziehungsweise Merkmale verarbeitet, somit nimmt die Komplexität Schicht für Schicht zu.

Für die gestellte Aufgabe scheint sich Deep-Learning anzubieten, da riesige Mengen an Bilddaten zur Verfügung stehen. Die Bewältigung dieser Bilddaten werden in dieser Arbeit von adversarialen Netzwerken übernommen, die durch Training an sehr vielen Bildern eigenständig lernen sollen.

Die vorliegende Bachelorarbeit gehört in den Bereich der Bildverarbeitung. In den dabei eingesetzten Deep-Learning-Algorithmen scheint viel Potential zu liegen für verschiedene Anwendungsgebiete, sowohl in der Industrie, der Wissenschaft, der Medizin und im Alltag.

Kapitel 2

Motivation und Zielsetzung

Bisher müssen bildliche Szenen recht aufwändig im Labor vorbereitet und erprobt werden, um dann von einem visuellen Sensor erfasst und aufgezeichnet zu werden.

Doch die Verfügbarkeit der für die Erfassung notwendigen Hardware kann durch verschiedene Faktoren limitiert sein. Dies können finanzielle Gründe sein, aber ebenso kann beispielsweise das hierfür notwendige parallele Arbeiten aus Kapazitäts- oder Zeitgründen nicht oder nur eingeschränkt möglich sein.

Wäre es stattdessen möglich, Szenen in einer virtuellen Umgebung darzustellen und diese mit einer möglichst genauen virtuellen Approximation realer Sensoren wahrzunehmen, könnten quasi unbegrenzt viele Anwender unabhängig voneinander und simultan mit der simulierten Hardware arbeiten. Algorithmen, welche auf bestimmte, sensorspezifische Daten ausgelegt und angewiesen sind, könnten somit auch mit Daten versorgt werden, welche in einer virtuellen Umgebung aufgenommen und anschließend durch ein trainiertes Modell verarbeitet wurden. Hier könnte also durchaus Potential stecken, Arbeitsabläufe flexibler, effizienter, schneller und daraus resultierend auch kostengünstiger zu gestalten.

Das Ziel beziehungsweise die Kernfrage dieser Arbeit besteht darin zu prüfen, ob es möglich ist, ein Modell zu trainieren, das aus gerenderten Szenen als Input möglichst realitätsnahe (oder sensorspezifische, also nah an der von einem Sensor wahrgenommenen Realität befindliche) Bilder als Output generiert.

Im Rahmen dieser Arbeit werden in Unreal Engine 4 aufgenommene Bilder als Input verwendet und versucht aus diesen Bildern mithilfe von Deep Learning ein Modell zu trainieren, das schlussendlich Bilder generiert, welche kaum oder im besten Fall gar nicht von „echten“ mit einer Kinect-Kamera aufgenommenen Bildern, zu unterscheiden sind.

Hierfür wird der Algorithmus CycleGAN[2] verwendet. Dieser Algorithmus trainiert unter Verwendung eines Bilddatensatzes der Herkunftsdomäne und eines Bilddatensatzes der Zieldomäne ein Modell, welches Bilder zwischen der Herkunftsdomäne und der Zieldomäne übersetzt und umgekehrt. Das trainierte Modell kann also Bilder bidirektional übersetzen. Dabei kann der verwendete generative Algorithmus durch das iterative Training selbst lernen, Unterscheidungsmerkmale zu identifizieren und mit wachsender Treffsicherheit zu arbeiten.

Da die von Deep-Learning-Algorithmen unter den oben beschriebenen Bedingungen trainierten Modelle nicht immer problemlos von Menschen nachvollziehbar sind, ist jeder Trainingsvorgang als eine Art Experiment zu verstehen, dessen Ausgang nicht zwingend deterministisch ist. Aus diesem Grund gilt es Experimente mit möglichst idealen Input-Datensätzen zu entwickeln, die zu den gewünschten Ergebnissen führen.

2.1. Abgrenzung und Einschränkungen

Alltag und Geschäftsleben werden immer mehr vom Deep Learning (DL) bestimmt. Deep Neural Networks (DNNs) machen viele Arbeitsschritte klassischer Netze überflüssig, da sie alle Zwischenschritte selbst übernehmen. DNNs müssen nur Daten (z.B. Bilder) präsentiert werden, wie diese zu identifizieren sind, finden sie eigenständig heraus. Da sie den geschichteten Aufbau hierarchischer Features abbilden können, sind sie für die Bildverarbeitung immens wichtig. Ihr volles Potential erhält die Netztiefe jedoch erst durch die Anpassung der Modellarchitektur an das vorliegende Problem. Die besondere Herausforderung besteht darin, Algorithmen auszuwählen, die geeignet sind, für vorliegende komplexe Probleme passende Lösungen zu finden. In der vorliegenden Arbeit geht es um Generative Adversarial Networks (GANs), also um eine Klasse generativer Modelle, die ausgehend von einer Zufallsstichprobe weitere Realisierungen (Samples) erzeugt, ohne vorher ein Wahrscheinlichkeitsmodell explizit aufzustellen. Die Experimente werden mit CycleGAN durchgeführt.

Kapitel 3

Related Works

In der vorliegenden Arbeit geht es um Generatives Deep Learning, also Modelle, welche aus Trainingsdaten Wahrscheinlichkeitsverteilungen erlernen mit der Möglichkeit aus der gelernten Verteilung neue Samples (mit ähnlicher Struktur) zu ziehen. Im Folgenden werden einige generative Modelle vorgestellt, die mehr oder weniger gut zur Bildgenerierung geeignet sind.

Zunächst werden autoregressive Modelle vorgestellt, die die Modellierung zum Sequenzproblem machen. Dann folgen Autoencoder beziehungsweise die in der Praxis oft zur “Datenkomprimierung “ eingesetzten Variational Autoencoders (VAEs)[3][4]. Im Vergleich zu VAEs sind Generative Adversarial Networks (GANs) zwar recht instabil und schwerer zu trainieren, dennoch scheint sich der erhöhte Aufwand zu lohnen. GANs sind nicht nur vielseitig einsetzbar, mit ihnen lassen sich auch neue (qualitativ hochwertige) Bilder erzeugen, die beispielsweise den Stil eines Bildes auf einen anderen Bildinhalt übertragen können. GANs können auch domänenübergreifend lernen, CycleGAN sogar mit ungepaarten Daten.

3.1. Autoregressive Modelle

Autoregressive Modelle sind dadurch gekennzeichnet, dass ihre Ausgabe von vorherigen Werten abhängt. Ein Bild kann als Pixelsequenz betrachtet werden. Zur Modellierung sequenzieller Daten werden oft tiefe neuronale Netze trainiert, um Pixelwerten in Bezug zu den vorherigen vorherzusagen. Die gemeinsame Wahrscheinlichkeit wird als Produkt bedingten Wahrscheinlichkeiten einzelner Pixel quantifiziert.

$$p(\mathbf{x}) = \prod_{i=1}^{n^2} p(x_i | x_1, \ldots, x_{i-1})$$

Im Folgenden werden Pixel RNN[5] und Pixel CNN[6] vorgestellt, die eine explizite Verteilung diskreter Pixelwerte (Pixelsequenzen) erlernen, aber mit unterschiedlichen neuronalen Netzen (NN).

3.1.1. PixelRNN

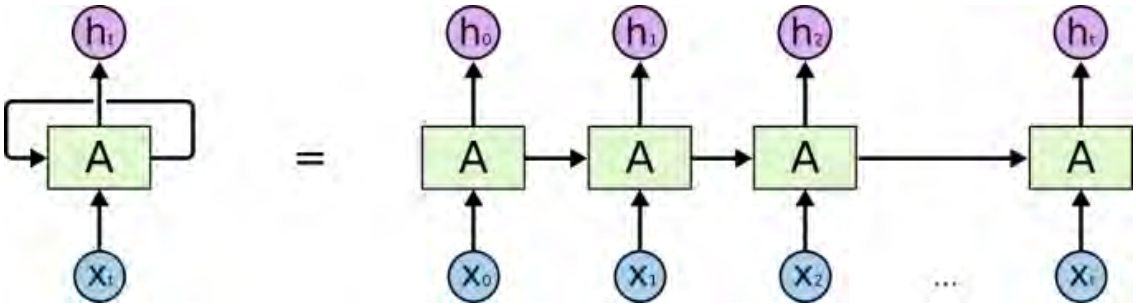
PixelRNN modelliert die bedingte Verteilung der Pixel mit rekurrenten Netzen. Eine Ecke wird als Startpunkt festgelegt. Die Vorhersage der Bildpixel erfolgt dann Pixel für Pixel nacheinander entlang der beiden räumlichen Dimensionen. Jeder Pixelwert wird von allen zuvor erzeugten Pixelwerten und von allen drei Farbkanälen (RGB) konditioniert.

In Feedforward Netzen (FFNs) laufen Daten bzw. Signale von Input- zu den Output-Units, also nur „vorwärts“. Rekurrente neuronale Netze (RNN) hingegen sind Feedback-Netze, Informationen werden nicht nur an die nächste Schicht weitergegeben, sondern auch rückgekoppelt. Dabei können Neuronen mit sich selbst (direct feedback), mit Neuronen derselben Schicht (lateral feedback) oder mit vorangegangenen Schichten (indirect feedback) verbunden sein. Durch diese Rückkopplungen (internen Schleifen) sind die Daten innerhalb des Netzes recurrent (wiederkehrend).

RNNs können Abhängigkeiten zwischen zeitlich versetzten Eingaben erkennen und den durch die Eingaben verursachte Zustand - ähnlich der menschlichen Kognition - in einer Art kurzes „Gedächtnis“ (Memory) speichern und später für Vorhersagen wieder heranziehen.

RNNs können über die zurückliegenden Zeitschritte entfaltet (Unfolded RNN) und dann wie FNNs durch Backpropagation trainiert werden (Backpropagation Through Time, BPTT).

Abbildung 3.1.: *Ein rekurrentes neuronales Netzwerk*[1]



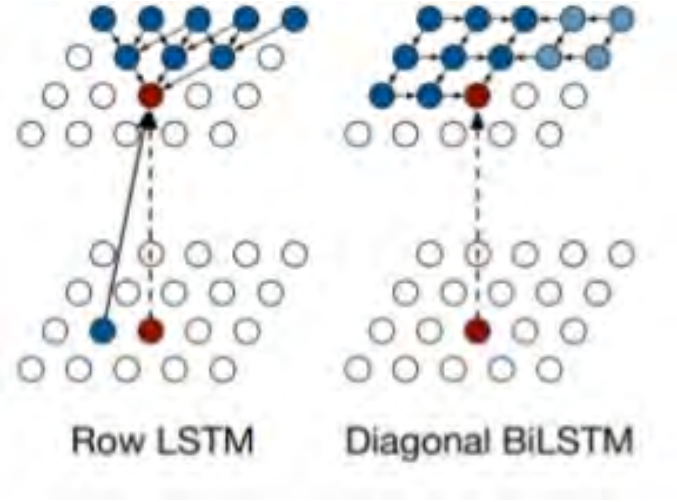
Für lange Sequenzen können RNNs als tiefe Netze (in Bezug auf die Anzahl der Schichten) verstanden werden, bei denen beim Trainieren (wie von tiefen FFNs bekannt) Vanishing Gradient auftritt. Um das Problem des verschwindenen Gradienten zu lösen, wurde ein erweitertes RNN, das Long Short-Term Memory (LSTM) [7], eingeführt und später noch weiterentwickelt [8]. Durch die Einführung von Input-, Forget- und Output-Gates kann der Informationsfluss reguliert und in einem “langen Kurzzeitgedächtnis” vor dem allmählichen Verschwinden geschützt werden.

Es gibt verschiedene 2-dimensionale LSTM-Varianten: Zeilen-LSTM verarbeitet ein Bild zeilenweise von oben nach unten für eine ganze Zeile auf einmal, erfasst wird dabei nur ein

[1] `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`

dreieckiger Bereich über dem Pixel (Triangular receptive field), Diagonal BiLSTM generiert durch Bewegungen in zwei Richtungen von der einen zur schräg gegenüberliegenden Ecke, so wird der gesamte Bereich (Full dependency field) erfasst. RNN Generierungen sind (durch das wiederholte Durchlaufen von Datensätzen) grundsätzlich langsam, Zeilen-LSTM etwas weniger langsam als Diagonal BiLSTM (best log likelihood).

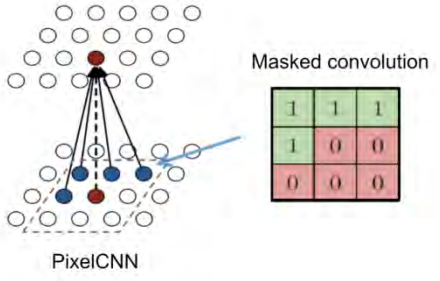
Abbildung 3.2.: *Unterschied in der Funktionsweise von Zeilen-LSTM und diagonalem LSTM*[2]



3.1.2. PixelCNN

PixelCNN[6] verwendet Convolutional Neural Networks (CNNs). Genauer gesagt: Mehrere Convolutional Layers, aber keine Subsampling-Layer (Ausgabe hat dieselbe Dimension wie die Eingabe). Die Verarbeitung von Bildern mit PixelCNN erfolgt wie bei PixelRNN sequentiell, aber nicht Pixel für Pixel, sondern in Regionen. Damit der zukünftige Kontext nicht gesehen wird, arbeitet PixelCNN mit Maskierungen (Nullsetzen).

Abbildung 3.3.: *Maskierung bei PixelCNN* [2]



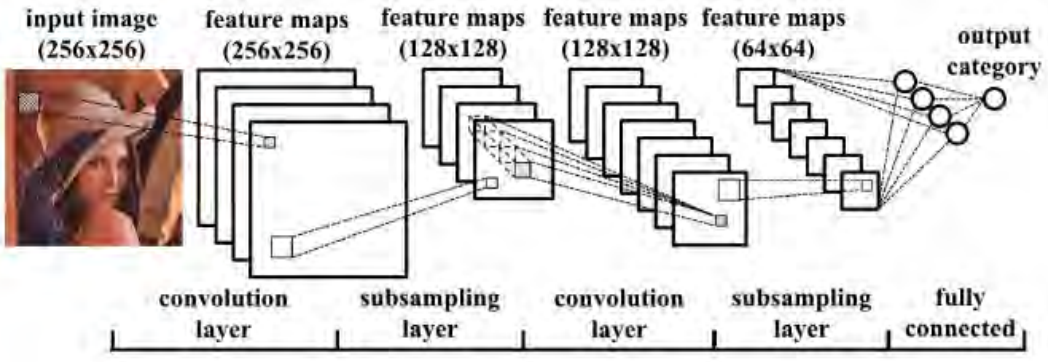
[2] abgewandelt von `https://www.slideshare.net/neouyghur/pixel-recurrent-neural-networks-73970786`

LSTM-Schichten erfassen Langzeitabhängigkeiten und berechnen sequentiell jeden Zustand (Empfangsfeld eher unbegrenzt), was mit einem hohen Rechenaufwand verbunden ist. Convolutional Layers (Faltungsschichten), die ein begrenztes Empfangsfeld erfassen und Merkmale für alle Pixelpositionen gleichzeitig berechnen, können dieses Problem lösen. Da PixelCNN nicht alle vorangegangenen Zeitpunkte und Durchgänge berücksichtigt, verläuft die Generierung im Vergleich zu PixelRNN weniger (aber immer noch) langsam, es entstehen aber blind spot Probleme.

Bei Multi-Layer Perceptrons (MLPs) sind die Neuronen der einzelnen Schichten vollständig verknüpft (Dense Layer), wodurch sich viele Parameter ergeben und somit die Gefahr von Redunanz besteht. Dense Layer erlernen aus Input(bild)daten globale Muster. Sollten diese Muster an einer anderen Position erneut vorkommen, müssen sie diese erneut lernen, wodurch sie sehr ineffizient sind. Im Unterschied dazu sind bei Convolutional Neural Networks (CNNs) nicht alle Neurone einer Schicht mit allen Neuronen der nächsten Schicht, sondern nur mit ausgewählten Neuronen der vorherigen Schicht verbunden. Dies spart viele Parameter und somit Verarbeitungsschritte, was tiefere Netze (im Sinne von Schichten) und größere Eingaben ermöglicht. CNNs arbeiten sehr effizient, denn sie können lokale Muster erlernen und diese an anderen Positionen wieder erkennen, darüber hinaus können mehrere Convolution Layers sogar räumliche Hierarchien eines Musters erlernen.

Vereinfacht ausgedrückt bestehen CNNs aus drei Layer-Typen: Input-, Convolution- und Output-Layers. CNNs transformieren Inputdaten, z.B. ein Bild (Matrixparameter: Höhe x Breite x Farbtiefe) stufenweise über Faltungsschichten in Output-Klassen. Die Convolution Layers stellen den Kern eines CNNs dar, sie falten Inputdaten mittels Convolution Kernels immer wieder neu (feature extraction). Das Ergebnis einer Eingabenfaltung mit ein und demselben Kernel wird feature map genannt. Für die räumliche Auflösung werden mehrere Convolution layer benötigt. Der Output Layer ist fully connected (softmax).

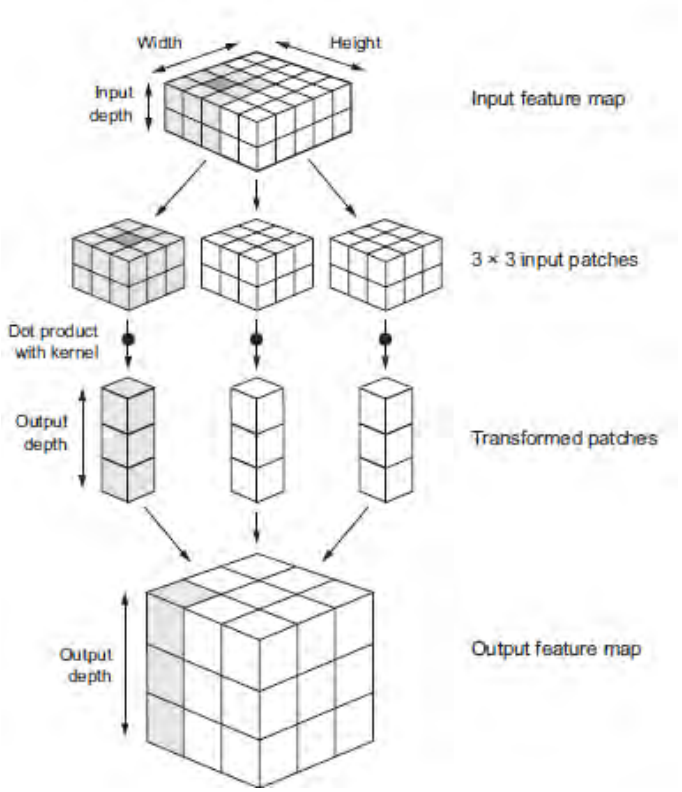
Abbildung 3.4.: *Faltungs- und Subsamplingschichten [9]*



Für die Faltungsoperationen werden Kernel bestimmter Größe (meist 3x3) und Schrittweite (meist 1) über die Eingangsdaten geschoben. Bei der Faltung gleiten die meist 3x3 Pixel großen Fenster schrittweise über die 3-D-Feature-Map der Eingabe und extrahieren an jeder möglichen Position einen 3-D-Patch der Merkmale dieser Umgebung. Die 3-D-Patches werden

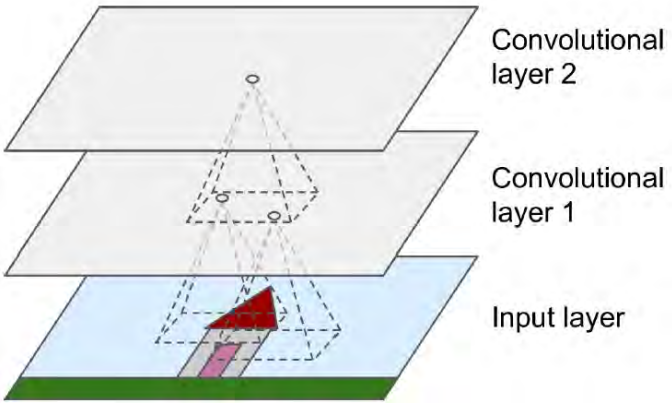
durch den Convolution Kernel in einen 1D-Vektor transformiert und zu einer 3-D-Ausgabe zusammengesetzt. Dabei sind die räumlichen Bereiche der Feature-Map der Eingabe zugeordnet (beispielsweise enthält die untere rechte Ecke der Ausgabe Informationen über die untere rechte Ecke der Eingabe).[10, p. 167]

Abbildung 3.5.: *Funktionsweise der Faltung[10, p.125]*



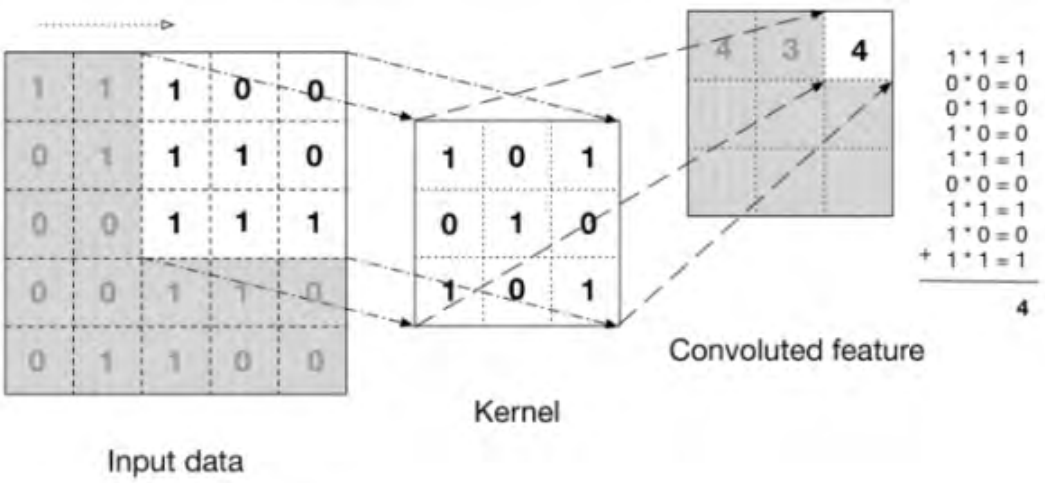
Wichtig für das Verstehen von Convolution Layers ist deren lokale Konnektivität. Nicht jeder Pixel des Input layer ist mit einem Neuron des ersten Convolution layers verbunden, auch die Neuronen des nächsten Convolution Layers sind nicht komplett, sondern nur mit solchen Neuronen des folgenden Convolution layers verbunden, die in einem Rechteck darunter lokalisiert sind.

Abbildung 3.6.: *Lokale Konnektivität von Convolution Layers [11, p.356]*



Mathematisch betrachtet: Die Inputdaten werden nacheinander mit dem Convolution Kernel multipliziert, die Summen addiert und dann in die Ausgabe (convoluted feature) geschrieben.

Abbildung 3.7.: *Anwendung eines Faltungskerns[12, p. 251]*



Autoregressive Modelle sind zwar leicht und stabil trainierbar, aber langsam. Die Generierung mit rekurrenten Netzen nimmt besonders viel Zeit in Anspruch (nacheinander wird jeder Zustand berechnet), liefert aber gute Samples (klar und kohärent). Durch die Verwendung von Faltungsschichten (parallele Berechnungen) kann die Generierung beschleunigt werden, bleibt jedoch (durch sequentielle Vorhersagen) langsam.

Die beiden oben vorgestellten autoregressiven Modelle generieren Bilder sequenziell. Dies ist anders bei Modellen mit latenten Variablen wie VAEs und GANs, die das ganze Bild auf einmal vorhersagen.

3.2. Autoencoder und Variational Autoencoders (VAE)

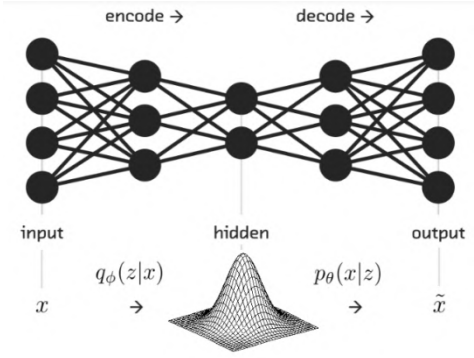
Die Idee eines Autoencoders (AE) ist schon relativ alt [13][14]. Ein Autoencoder ist ein typischer Vertreter der unüberwachten Lernverfahren, der eine explizite Verteilung aus einem latenten Raum (naturgemäß einem Vektorraum)erlernt.

Jeder Datenpunkt eines hoch-dimensionalen Bildes besteht aus vielen Merkmalen, von denen oft wenige ausreichen, um den kompletten Datensatz approximativ gut zu charakterisieren.

Ein Autoencoder-Netz besteht aus zwei miteinander verbundenen FFNs, dem Encoder und dem Decoder. Für die Verarbeitung von Bildern bietet sich als Encoder ein CNN an und der Decoder übernimmt dann die analoge Entfaltung.

Der Encoder komprimiert Eingabedaten in einen latenten Raum, aus dem dann der Decoder die Eingabedaten rekonstruiert.

Abbildung 3.8.: *Komprimiertes hidden layer*[3]



Es geht ausdrücklich nicht um “perfektes Kopieren”,sondern der Autoencoder soll selbst effiziente Codierungen erlernen:

- Der Encoder reduziert die Dimensionen (Anzahl an Neuronen) immer weiter, bis in der Netzmitte die kleinste Schicht erreicht wird.
- Der hidden code (“Bottleneck”) dient nicht nur der komprimierten Repräsentation der Eingabe, sondern stellt gleichzeitig die Eingabeschicht des Decoders dar.
- Der Decoder baut die Dimensionen Stück für Stück wieder auf (bis zum Erreichen derselben Dimensionen wie die ursprüngliche Eingabe).

[3] `https://towardsdatascience.com/what-the-heck-are-vae-gans-17b86023588a`

Ein Autoencoder wird trainiert, indem derselbe Eingang und Ausgang zugeführt wird. Das Netz soll lernen die Eingabe so genau wie möglich nachzubilden.

Durch Einschränkungen der Codierung, also der Ausgabe des Encoders, kann der Autoencoder dazu gebracht werden, latente Repräsentationen der Daten zu erlernen.

- Am gebräuchlichsten ist die Dimensionsreduktion, bei der die Anzahl der hidden layer niedriger ist als die Größe der Ein-und Ausgabe. Sie dient der feature extraction.

- Der Autoencoder sollte nicht zu viele Daten erhalten, denn zu viele Punkte aus dem latenten Raum führen nicht zu in sich stimmigen Bildern [15].

Das Konvergieren kann beschleunigt werden, indem anstelle der quadratischen Abweichung ein Rekonstruktionsverlust verwendet wird. Hierfür ist die Kreuzentropie besonders geeignet [11, p. 140]. Das Training des Netzes kann nicht nur durch Backpropagation, sondern auch durch Rezirkulation erfolgen[16]

Verfügen Encoder und Decoder über eine gewisse Tiefe, so können experimentell im Vergleich zu flachen oder linearen Autoencodern bessere Komprimierungen erreicht werden[17].

Es existieren verschiedene Varianten von Autoencodern. Ein Denoising-Autoencoder (DAE) [18] erhält als Eingabe einen beschädigten Datenpunkt und wird darauf trainiert, den ursprünglichen, nicht beschädigten Datenpunkt zu rekonstruieren. Zur Generierung neuer (Bild)daten oder um Variationen in ein Eingabebild zu bringen sind Standard Autoencoder nicht geeignet, da sie weder ausreichend strukturierte noch kontinuierliche Räume liefern. In der Praxis werden sie daher, abgesehen vom Entrauschen, kaum noch verwendet.

3.2.1. Variational Auto(en)coder (VAE)

Ein VAE [3][4] ist eine moderne Version eines AEs, die konstruktionsbedingt kontinuierliche, gut strukturierte latente Räume liefert. Die dahinter stehende Idee ist, einen Encoder so zu trainieren, dass er auf Bayes'sche Schlussfolgerungen zurückgreift. Der VAE-Encoder (approximatives Inferenznetz) legt ein Eingabebild nicht als komprimierten festgelegten Code im latenten Raum ab, sondern wandelt das Eingabebild in zwei stochastische Parameter um. Er gibt nicht nur einem, sondern gleich zwei Kodierungsvektoren aus, einen Vektor der Mittelwerte μ und einen Vektor der Varianz σ^2.

Ein VAE ist ein probabilistischer Autoencoder. Einem Bild werden zwei Vektoren zugeordnet um die Wahrscheinlichkeitsverteilung für den latenten Raum zu definieren, aus dem zufällige Punkte gezogen und dann rekonstruiert werden.

Die Parameter eines VAE werden über zwei Verlustfunktionen trainiert:

- einem Rekonstruktionsverlust (wie bei Standard Autoencodern), der die Abweichungen zwischen den decodierten Samples und der ursprünglichen Eingabe minimiert.

- einem Regularisierungsverlust (Minimierung der Kullbach-Leibler-Divergenz, also Optimieren der Wahrscheinlichkeitsverteilungsparameter), der das Erlernen von wohlgeformten latenten Räumen erleichtern und Overfittung verringern soll.

Um Backpropagation zu ermöglichen, muss ein Trick angewendet werden, die Umparametrierung. Anstelle der zufälligen Abtastung z N (ϕ, σ2I) wird zufälliges Rauschen ϵ N (0, I) erzeugt. Dadurch kann z deterministisch bezüglich ϕ und σ^2 abgetastet werden.

Abbildung 3.9.: *Sampling beim Autoencoder*[4]

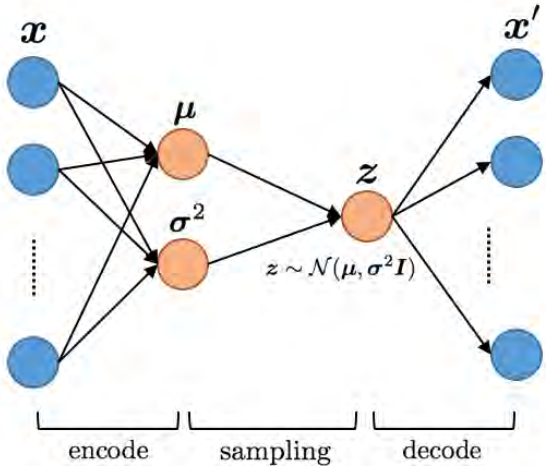
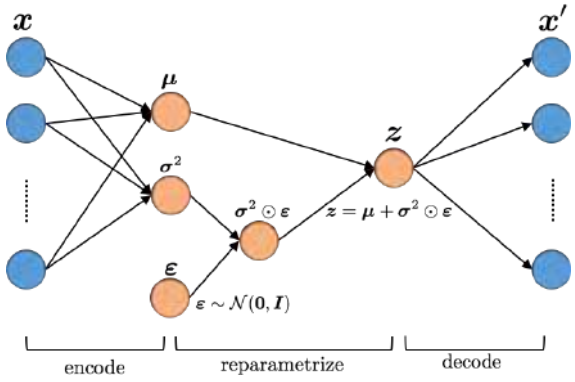


Abbildung 3.10.: *Umparametrierung*[4]



Da ϵ zufällige Punkte enthält, wird durch diese Vorgehensweise gewährleistet, dass Punkte, die im latenten Raum nah beieinanderliegen nach der Decodierung sehr ähnliche Bilder ergeben. Diese Stetigkeit zusammen mit einem niedrig dimensionalen latenten Raum erzwingen,

[4] `https://www.renom.jp/notebooks/tutorial/generative-model/VAE/notebook.html`

dass alle Richtungen im latenten Raum Achsen darstellen, die für die Variationen der Daten von Bedeutung sind. [10]

Ein VAE kann sehr gut für Änderung von bereits vorhandenen Daten verwendet werden. Bildern von Gesichtern kann z.B. eine Brille zugefügt oder entfernt werden, ein männliches kann in ein weibliches Gesicht umgewandelt werden oder umgekehrt. Allerdings lernt der VAE-Encoder (Inferenznetz) nur für ein Problem, also von einem x auf z zu schließen. Diese Datenspezifität führt zu Problemen in der Bildverarbeitung. Ein auf menschliche Gesichter trainierter Encoder hat Probleme mit Gebäuden oder anderen Objekten. (keine „Allzweckkomprimierung“).

VAEs können zwar neue Bilder generieren, von großem Nachteil ist jedoch, dass VAE generierte Bilder oft unscharf sind. Die Ursachen dafür sind bisher noch weitestgehend unbekannt.

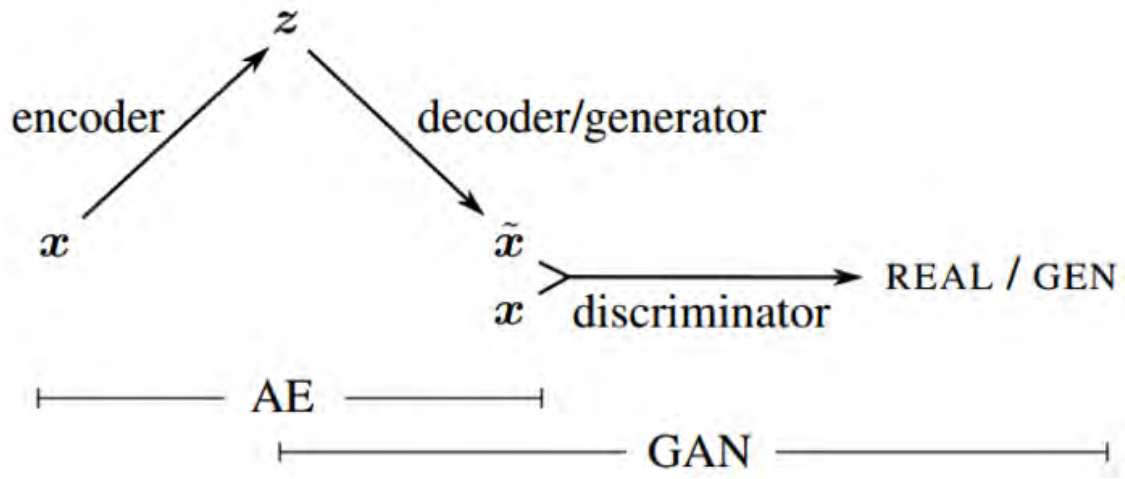
Im Vergleich dazu, liefern GANs zwar keine zusammenhängenden klar strukturierten latenten Räume, ermöglichen aber die Erzeugung fotorealistischer Bilder von (meist) hoher Qualität.

3.2.2. VAE-GAN

GANs bestehen aus zwei kontradiktorischen Netzen: Ein Generator (G) erzeugt aus einem Zufallsvektor Bilder und ein Diskriminator (D) versucht, die generierten von realen Bildern aus den Trainingsdaten zu unterscheiden. Im Laufe des Trainings liefert G zunehmend bessere Bilddaten, um D zu täuschen. D hingegen lernt echte immer sicherer von generierten Bildern zu unterscheiden, womit er G zwingt, bessere Bilder zu generieren. Schließlich sollen Bilder generiert werden, die selbst für Menschen nicht von realen Bildern aus den Trainingsdaten zu unterscheiden sind.

VAEs [3][4] neigen dazu, während der Rekonstruktionsphase verschwommene Ausgaben zu erzeugen. Um dieses Problem zu lösen, wird bei einem VAE-GAN [19] anstelle eines VAE-Decoders ein GAN-Diskriminator verwendet, um die Verlustfunktion zu erlernen.

Abbildung 3.11.: *VAE-GAN Architektur: Der Diskriminator des GAN erhält seinen Input vom Decoder des VAE*[5]



[5] https://towardsdatascience.com/what-the-heck-are-vae-gans-17b86023588a

Kapitel 4

Verwendete Software

4.1. CycleGAN

`https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix`

CycleGAN ist ein Algorithmus zur domänen-übergreifenden Bild-zu-Bild-Übersetzung von ungepaarten Datensätzen. An die Stelle eines gepaarten Bildes in der Zieldomäne tritt bei CycleGAN eine zweistufige Transformation des Quellbildes

4.2. Unreal Engine 4

`https://www.unrealengine.com/`

Unreal Engine 4 (UE4) ist eine Spieleengine, welche im Rahmen dieser Arbeit hauptsächlich für das Rendern realitätsnaher, virtueller Szenen in Echtzeit verwendet wurde.

4.2.1. URoboVision

`https://github.com/robcog-iai/URoboVision/tree/guan`

URoboVision ist ein Plugin für UE4, welches Farb- und Tiefenbilder in Unreal aufnimmt und dann entweder als herkömmliche Bilddatei (z.B. JPEG), lokal als BSON-Datei oder auf einer remote Mongo-Datenbank speichert. Im Rahmen dieser Arbeit wurde das URoboVision Plugin in zweierlei Hinsicht modifiziert. Zum einen wurde die Kompabilität mit neueren UE4-Versionen hergestellt, zum anderen wurde ein zusätzliches Feature implementiert, welches es erlaubt die aufgenommenen Bilder zur Laufzeit via UROSBridge direkt an ein ROS-System im Netzwerk zu schicken.

4.2.2. UROSBridge

`https://github.com/robcog-iai/UROSBridge`

UROSBridge ist ein PlugIn für UE4, welches die Kommunikation zwischen UE4 und eines ROS-System mit rosbridge über Websockets ermöglicht. Das PlugIn unterstützt sowohl das Abonnieren als auch das Veröffentlichen von ROS-Nachrichten (topics).

4.2.3. libmongo

`https://github.com/robcog-iai/libmongo`

libmongo ist ein Plugin für UE4, welches MongoDB-Treiber in UE4 zur Verfügung stellt. Von RoboSherlock aufgenommene Szenen werden in einer MongoDB gespeichert und dann im Rahmen dieser Arbeit mithilfe von libmongo in Unreal repliziert.

4.3. ROS

4.3.1. RoboSherlock

`https://github.com/RoboSherlock/robosherlock`

RoboSherlock ist ein Framework für cognitive perception, basierend auf unstrukturiertem Informationsmanagement (unstructured information management, UIM). Im Rahmen dieser Arbeit wurden reale Szenen aus von RoboSherlock erkannten Objekten in UE4 repliziert.

4.3.2. rosbridge

`http://wiki.ros.org/rosbridge_suite`

rosbridge ist das Pendant zu UROSBridge auf der ROS-Seite. rosbridge stellt ein JSON-API zur Verfügung, welches von UROSBridge genutzt wird, um ROS-Funktionailität in nicht-ROS-Programmen zu ermöglichen.

Kapitel 5

Grundlagen

5.1. Funktionsweise von GANs im Allgemeinen

Generative Modelle existieren schon länger, GANs wurden jedoch erstmals 2014 von einer von Ian J. Goodfellow geleiteten Forschergruppe im Paper „Generative Adversarial Nets“ eingeführt [1]. Die Bezeichnung GAN steht für *generative adversarial network*. Es handelt sich dabei um eine neuronale Netzstruktur, welche aus zwei gegeneinander arbeitenden (daher adversarial) Netzen besteht, welche konfligierende Ziele verfolgen.

Die beiden gegnerischen Netzstrukturen sind:

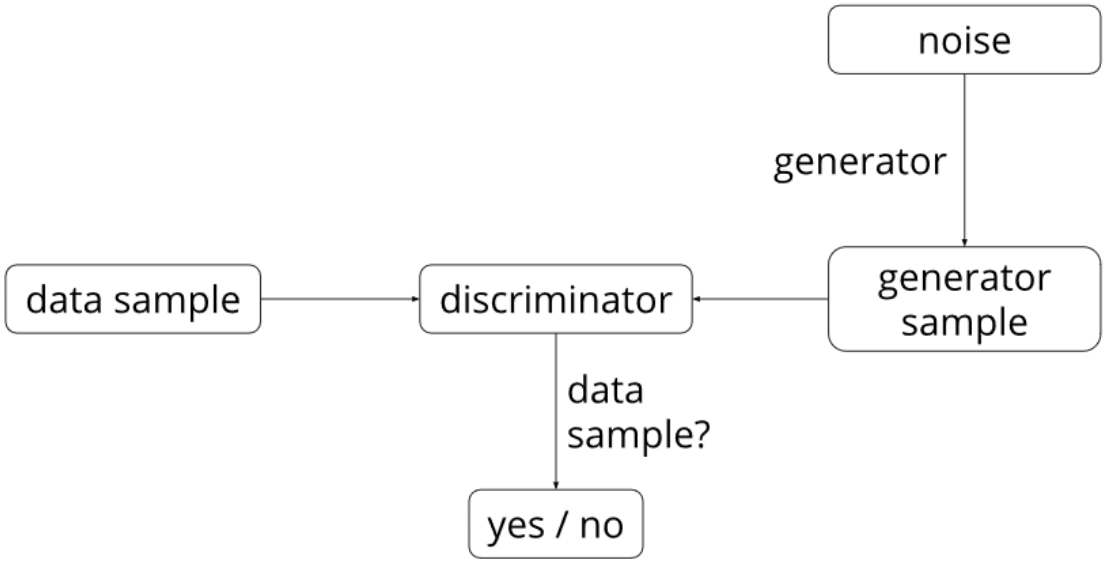
- Ein generatives Modell G, welches aus dem Trainings-Datensatz neue Daten abstrahiert. Dies ist der sogenannte „Generator“.

- Ein diskriminatives Modell D, welches ermittelt, wie wahrscheinlich es ist, dass eine Stichprobe aus den ursprünglichen Trainingsdaten stammt statt aus G. Dies ist der sogenannte „Diskriminator“.

Generator und Diskriminator trainieren sich simultan gegenseitig und lernen dabei aus den Resultaten des jeweils anderen voneinander: Aus dem Rauschen von Inputdaten (z) erzeugt der Generator neue Daten ($G(z)$), die anschließend vom Diskriminator auf Authentizität untersucht werden. Dazu überprüft der Diskriminator die generierten Daten auf die Wahrscheinlichkeit, mit der sie aus dem ursprünglichen Datensatz stammen ($D(G(z))$). Basierend auf dieser Wahrscheinlichkeit optimiert der Generator sein Vorgehen schrittweise, bis der Diskriminator generierte Daten nur noch schwer (oder am besten gar nicht) von echten Daten unterscheiden kann. Gleichzeitig verbessert auch der Diskriminator seine Zuverlässigkeit, um möglichst viele (am besten alle) Fälschungen zu enttarnen.

Im Trainingsverlauf werden die Konkurrenten immer besser. Dadurch dass der Diskriminator lernt, generierte Daten besser von realen Daten zu unterscheiden, ist der Generator gezwungen, realistischere Daten zu erzeugen um den Diskriminator zu täuschen.

Abbildung 5.1.: *Grob schematische Darstellung der Funktionsweise eines GANs*[1]



Diese Beziehung lässt sich formal als Min-Max-Zwei-Spieler-Spiel verstehen, bei dem G versucht D mit „gefälschten" Daten zu täuschen, während D parallel dazu versucht immer besser darin zu werden die gefälschten Daten von echten Daten zu unterscheiden. [1]

D und G spielen also folgendes Zwei-Spieler-Spiel mit *value function* $V(G, D)$:

$$\min_G \max_D V(G,D) = \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})} \underbrace{[\log D(x)]}_{\substack{\text{Diskriminator-}\\ \text{Output für reale}\\ \text{Daten } x}} + \mathbb{E}_{z\sim p_z(z)}[\log(1 - \underbrace{D(G(z))}_{\substack{\text{Diskriminator-}\\ \text{Output für generierte}\\ \text{Daten } G(z)}})]$$

Wobei $p_z(z)$ das Rauschen der Inputdaten und $\mathbf{x}$ die realen Daten sind. Dann repräsentiert $G(z; \theta_g)$ die Abbildung in den Datenraum, wobei G eine differenzierbare Funktion ist, welche durch das multilayer perceptron (MLP) mit den Parametern θ_g definiert wird. Als zweites MLP wird $D(x; \theta_d)$ definiert, welches einen einzelnen Skalar ausgibt.

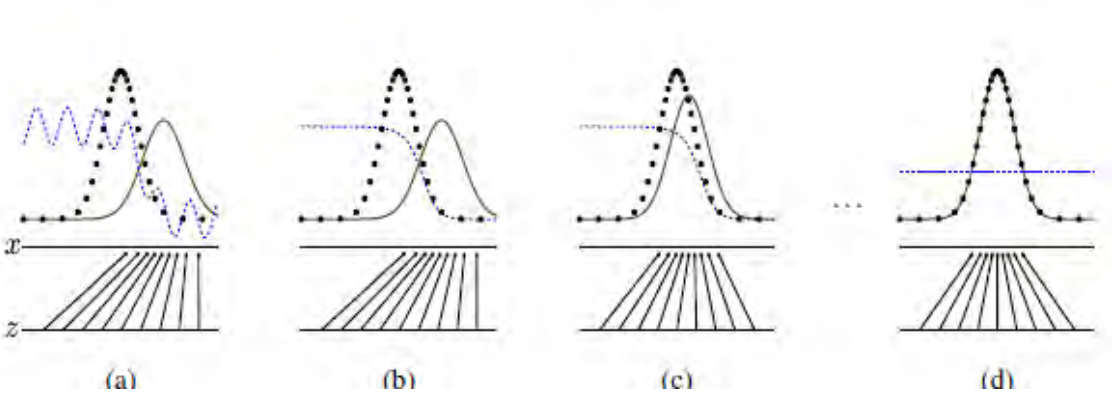
Der erste Summand beschreibt den Output für reale Daten $\mathbf{x}$.

Der zweite Summand gibt an, wie gut der Diskriminator generierte Daten erkennt ($D(G(z))$) und wie gut die vom Generator gefälschten Daten ($G(z)$) sind.

D wird trainiert, sodass die Wahrscheinlichkeit maximiert wird, sowohl Trainingsdaten als auch von G generierte Daten korrekt einzuteilen. Gleichzeitig wird G trainiert, sodass $\log(1 - D(G(z)))$ minimiert wird.

[1] `https://ishmaelbelghazi.github.io/ALI/`

Abbildung 5.2.: *Trainingsstadien in Richtung Nash-Equilibrium [1]*



(a) Vor oder zu Beginn des Trainings kann der Diskriminator generierte Daten noch gut erkennen, da diese sich deutlich von den Trainingsdaten unterscheiden.

(b) Nachdem der Diskriminator trainiert wurde, nehmen die Schwankungen hinsichtlich der Sicherheit ab mit der D generierte Daten detektiert.

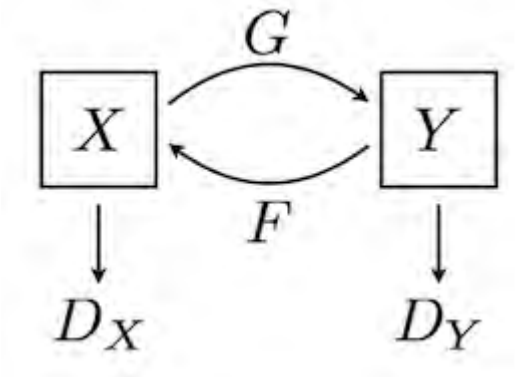
(c) Nachdem der Generator trainiert wurde, gleicht sich die Verteilung der generierten Daten der Verteilung der echten Daten an.

(d) Verteilung der generierten Daten quasi identisch zur Verteilung der generierten Daten, somit kann der Diskriminator nicht mehr zwischen echten von gefälschten Daten unterscheiden.

5.2. Funktionsweise von CycleGAN im Speziellen

CycleGAN[2] steht für *cycle consistent adversarial network*. Dieser Algorithmus wurde zur domänen-übergreifenden Bild-zu-Bild-Übersetzung von ungepaarten Datensätzen entwickelt [2]. Die Tatsache, dass keine gepaarten Datensätze erforderlich sind, vereinfacht das Sammeln oder Erzeugen von Datensätzen erheblich. An die Stelle eines gepaarten Bildes in der Zieldomäne tritt bei CycleGAN eine zweistufige Transformation des Quellbildes.

Abbildung 5.3.: *Grundlegende Architektur von CycleGAN [2]*



CycleGAN erweitert das Konzept des GANs um ein zweites GAN, insgesamt werden also vier Netze trainiert: Zwei Generatoren G, F, welche Bilder aus der Domäne **X** beziehungsweise **Y** generieren und zwei Diskriminatoren D_Y, D_X, welche zwischen erzeugten und ursprünglichen Bildern unterscheiden. Ein GAN besteht also aus dem adversarialen Paar (G, D_Y), das andere aus dem adversarialen Paar (F, D_X)

Die hinter *cycle consistency* stehende Idee ist, dass beim Übersetzen eines Bildes x von Domäne **X** in Domäne **Y** und anschließender Übersetzung des erzeugten Bildes zurück in Domäne **X**, das Ergebnis möglichst wie das Ursprungsbild x aussehen sollte. Analoges gilt für das Übersetzen eines Bildes y aus Domäne **Y**.

Für zwei ungepaarte Bilderdatensätze **X**, **Y** liefert CycleGAN eine Abbildung G mit $G : X \rightarrow Y$ und eine Abbildung F mit $F : Y \rightarrow X$. Um diese Abbildungen zu erlernen wird, analog zu einem herkömmlichen GAN, ein Ziel gesetzt:

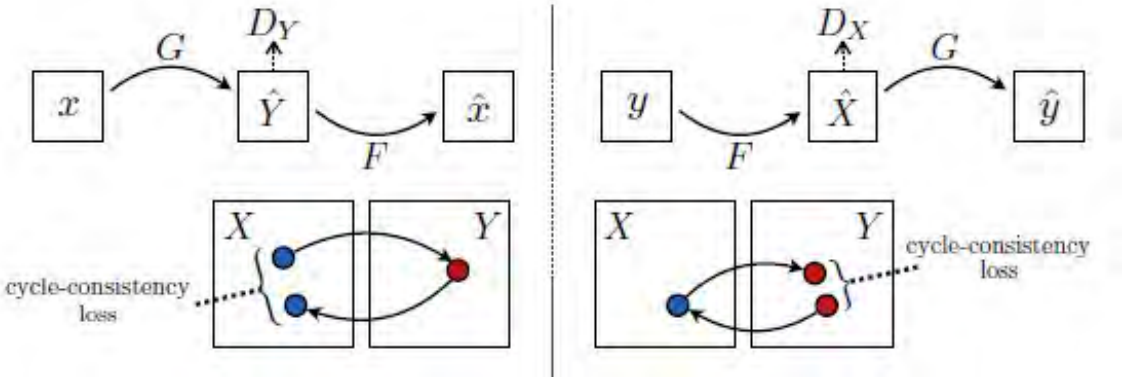
- *adversarial loss*: Sei $\widehat{y} \sim P(\widehat{Y}) = G(\mathbf{x}; \theta_g)$ ein Bild aus dem Generator G mit der Verteilung $P(\widehat{Y})$ und $y \sim P(Y)$ ein echtes Bild mit der Verteilung $P(Y)$. Als *adversarial loss* wird dann die Differenz zwischen $P(\widehat{Y})$ und $P(Y)$ bezeichnet. Ziel eines jeden GANs ist es also, diese Differenz zu minimieren, dass also ein generiertes Bild ununterscheidbar von einem echten Bild ist.

Bei ungepaarten Datensätzen gibt es aber viele Parameter θ_g welche möglicherweise die Differenz der Verteilungen minimieren könnten. Die Chance eine Abbildung G zu erlernen, welche sinnhafte Verbindungen zwischen der Inputdomäne X und der Outputdomäne Y herstellt ist also sehr gering. Würde also nur die Minimierung des *adversarial loss* als Ziel verwendet, so wären die Resultate des Trainings weitestgehend bedeutungslos. es würden zwar realistische Bilder aus der Zieldomäne erzeugt werden, die domänenübergreifende Verknüpfung bliebe aber aus.

Um die Wahrscheinlichkeit sinnhafte Verbindungen zu erlernen zu maximieren wird komplementär eine zweite Art von *loss* eingeführt, um die beiden Domänen möglcihst eng miteinander zur verknüpfen. Diese zweite Art von *loss* ist eine Neuerung zum herkömmlichen GAN:

- *cycle consistency loss*: Der Zykluskonsistenzverlust wird berechnet, indem die rekonstruierten Bilder mit dem entsprechenden Eingabebildern verglichen werden. Der *cycle consistency loss* ergibt sich dann aus der Differenz zwischen $G(F(\mathbf{y}))$ und $\mathbf{y}$ sowie die Differenz zwischen $F(G(\mathbf{x}))$ und $\mathbf{x}$. Intuintiv lässt sich also sagen, dass der *cycle consistency loss* die Differenz zwischen einem Ursprungsbild $\mathbf{x}$ aus der Domäne X und dem Bild, welches entsteht wenn $\mathbf{x}$ zuerst nach Y übersetzt und dann zurückübersetzt wird. (Und analog für ein Bild $\mathbf{y}$ aus der Domäne Y). Es werden also „Rekonstruktionsfehler“, die bei der zweistufigen Transformation des Quellbildes unterlaufen gemessen.

Abbildung 5.4.: *Cycle consistency loss [2]*



In Abbildung links handelt es sich hierbei um sogenannten forward cycle consistency loss, also $\mathbf{x} \rightarrow G(\mathbf{x}) \rightarrow F(G(\mathbf{x})) \approx \mathbf{x}$. In Abbildung rechts um backwards cycle consistency loss, also $\mathbf{y} \rightarrow F\mathbf{y}) \rightarrow G(F(\mathbf{y})) \approx \mathbf{y}$

Kapitel 6

Umsetzung

6.1. UVisionLogger

Im Rahmen dieser Bachelorarbeit wurde ein Plugin für UE4 geschrieben, welches ermöglicht Farbbilder (RGB) und Tiefenbilder, sowie Bilder der Normalenvektoren von Flächen und Bilder mit Objektmaske in UnrealEngine aufzunehmen[1]. Diese Bilder können dann sowohl zur Laufzeit lokal in herkömmlichen Bildformaten, als auch als bson-Datei oder in einer Mongo-Datenbank im Netzwerk gespeichert werden. Zudem besteht die Möglichkeit aufgenommene Bilder zur Laufzeit via UROSBridge an ein ROS-System im Netzwerk zu senden, wo sie dann (von einem anderen, ROS-kompatiblen Programm verarbeitet werden können).

Dieses Plugin baut dabei auf zwei Versionen eines ähnlichen, älteren Plugins[2] [3] auf.

Besonders hervorzuheben ist hier die modifizierte `ReadPixels`-Funktion. `ReadPixels` schreibt die angezeigten Pixel des Viewports in den Farbbuffer `OutImageData`. Diese Funktion stammt ursprünglich aus `Runtime/Engine/Public/UnrealClient.h`, lässt sich ohne Modifikationen jedoch nur aus dem Renderthread heraus aufrufen. Ruft man sie jedoch aus dem Renderthread heraus auf, so muss der gesamte Renderthread auf die Ausführung der Funktion warten, was für massive FPS-Einbrüche sorgt.

[1] `https://gitlab.informatik.uni-bremen.de/jcmack/uvisionlogger`
[2] `https://github.com/robcog-iai/URoboVision`
[3] `https://github.com/robcog-iai/URoboVision/tree/guan`

```
void AUVCamera::ReadPixels(FTextureRenderTargetResource & RenderResource, ↩
    TArray<FColor>& OutImageData, FReadSurfaceDataFlags InFlags, FIntRect ↩
    InRect)
{

   // Read the render target surface data back.
   if (InRect == FIntRect(0, 0, 0, 0))
   {
      InRect = FIntRect(0, 0, RenderResource->GetSizeXY().X, RenderResource↩
          ->GetSizeXY().Y);
   }
   struct FReadSurfaceContext
   {
      FRenderTarget  SrcRenderTarget;
      TArray<FColor> OutData;
      FIntRect Rect;
      FReadSurfaceDataFlags Flags;
   };

   OutImageData.Reset();
   FReadSurfaceContext ReadSurfaceContext =
   {
      RenderResource,
      &OutImageData,
      InRect,
      InFlags,
   };

   ENQUEUE_UNIQUE_RENDER_COMMAND_ONEPARAMETER(
      ReadSurfaceCommand,
      FReadSurfaceContext, Context, ReadSurfaceContext,
      {
         RHICmdList.ReadSurfaceData(
            Context.SrcRenderTarget->GetRenderTargetTexture(),
            Context.Rect,
             Context.OutData,
            Context.Flags
         );
      });
}
```

Kapitel 7

Reproduzierte Modelle

Die oben gezeigten Bilder sind das Resultat des Versuchs der Reproduktion der Ergebnisse aus „Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks“[2]. Die verwendeten Modelle wurden hierfür mit exakt den gleichen Parametern und exakt den gleichen Trainingsdatensätzen trainiert. Die trainierten Modelle wurden dann verwendet, um Bilder in den Stil der genannten Künstler zu transformieren. Dabei ist wichtig, dass die hierfür verwendeten Bilder zwar in der Art den Bildern aus dem Trainingsdatensatz ähneln, darin aber nicht selbst vorkommen. Die Modelle wurden also nicht anhand der gleichen Daten getestet, mit denen sie auch trainiert wurden.

Insgesamt sind die Ergebnisse hier vergleichbar mit denen aus dem genannten Paper. Es ist jedoch zu bedenken, dass es sich bei den im Paper gezeigten Datenpunkten vermutlich um besonders gute Ergebnisse handelt und „schlechtere“ Ergebnisse eher nicht gezeigt wurden.

Auffälligkeiten bei diesem Experiment:

- In einigen der generierten Bilder treten rasterartige Muster auf, welche so im Originalbild nicht zu finden sind. Diese Muster scheinen sich besonders an Stellen zu zu häufen, die im Ursprungsbild weitestgehend monochrom sind. Diese Artefakte sind ein bekanntes Problem beim Training mit CycleGANs[1], welches durch längeres Training oder größere Trainingsdatensätze lösbar sein könnte. Diese Anpassungen wurden hier jedoch nicht vorgenommen, da sich dieser Trainingsvorgang so nah wie möglich am Vorgang aus dem Paper orientieren sollte.

[1] `https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix/issues/518`

- Wie bei vielen Deep-Learning-Modellen tritt auch bei CycleGAN ein gewisses Maß an Overfitting ein. Dabei erkennt das Modell Strukturen, welche zwar im Trainingsdatensatz gehäuft vorkommen, allerdings im gegebenen Bild (zumindest für Menschenaugen) nicht wahrnehmbar sind und stellt diese überzeichnet dar. So enthalten die zum Training benutzten Bilder von Paul Cézanne überwiegend Porträts und Stilleben, während die zur übersetzenden Bilder hauptsächlich Landschaftsbilder sind. In einigen der übersetzen Bilder lassen sich also Stillleben- oder Porträt-artige Strukturen wiedererkennen.

- Das umgekehrte Problem kann ebenfalls auftreten: Enthält der Trainingsdatensatz nicht genug Bilder einer bestimmten Art, so kann CycleGAN Bilder dieser Art auch nur kaum verändern. Teils versucht das Modell zum Beispiel Bilder von Porträts in Landschaftsbilder zu übersetzen oder ähnliches. Dabei kann es jedoch kaum zu sinnhaften Verknüpfungen kommen.

Weiter wurden die eben trainierten Modelle dann zu Demonstrationszwecken dazu eingesetzt, Bilder aus der RobCoG-Küche in den Stil der jeweiligen Künster zu übersetzen.

Kapitel 8

Eigens trainierte Modelle

Für die folgenden Experimente wurden zwei verschiedene Datensätze von realen, mit einer Kinect-Kamera aufgenommenen Bildern verwendet. Dabei handelt es sich um einen für „Category Modeling from just a Single Labeling: Use Depth Information to Guide the Learning of 2D Models“ [20][1] angefertigten Datensatz und um den Datensatz ViDRILO[21]. Da ViDRILO sehr umfangreich ist (etwa 22,000 Bilder) wurde für die Experimente, eine zufällig ausgewählte Teilmenge der Bilder verwendet um die Dauer der einzelnen Experimente überschaubar zu halten. Für jedes Experiment wurden nur Bilder aus einem der realen Datensätze genutzt – die Datensätze wurden also nie vermischt.

Jedes Experiment lief jeweils die gleiche Anzahl an Trainingsgenerationen (200 „epochs“), was jedoch (abhängig unter Anderem von der Größe der Trainingsdatensätze) unterschiedlich lange reale Zeiträume bedeutet. Der Inhalt der jeweiligen Datensätze und Trainingsdurchläufe ist im Anhang aufgeschlüsselt (A.1, A.2)

Die Nomenklatur von CycleGAN[2] wird hier beibehalten, das heißt es gilt:

$real\_A = A$, $real\_B = B$

$fake\_B = F(A)$, $fake\_A = G(B)$

$rec\_A = G(F(A))$, $rec\_B = F(G(B))$

$idt\_A = F(B)$, $idt\_B = G(A)$

[1] `https://sites.google.com/site/quanshizhang/dataset`

8.1. unreal2kinect

Für dieses Experiment wurde der Datensatz von Quanshi Zhang et al.[20] verwendet. Der Datensatz mit virtuellen Szenen stammt aus der RobCoG-Küche[22][23][2] und wurde vom Autor dieser Arbeit aufgenommen.

Abb. 8.1.: *real_B_012*

Abb. 8.2.: *fake_A_012*

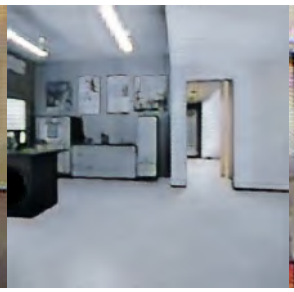


Abb. 8.3.: *rec_B_012*

Abb. 8.4.: *idt_B_012*

Abb. 8.5.: *real_B_009*

Abb. 8.6.: *fake_A_009*

Abb. 8.7.: *rec_B_009*

Abb. 8.8.: *idt_B_009*

Abb. 8.9.: *real_B_143*

Abb. 8.10.: *fake_A_143*

Abb. 8.11.: *rec_B_143*

Abb. 8.12.: *idt_B_143*

[2] `https://github.com/robcog-iai/RobCoG`

Abb. 8.13.: *real_B_200* ***Abb. 8.14.:*** *fake_A_200* ***Abb. 8.15.:*** *rec_B_200* ***Abb. 8.16.:*** *idt_B_200*

Abb. 8.17.: *real_B_174* ***Abb. 8.18.:*** *fake_A_174* ***Abb. 8.19.:*** *rec_B_174* ***Abb. 8.20.:*** *idt_B_174*

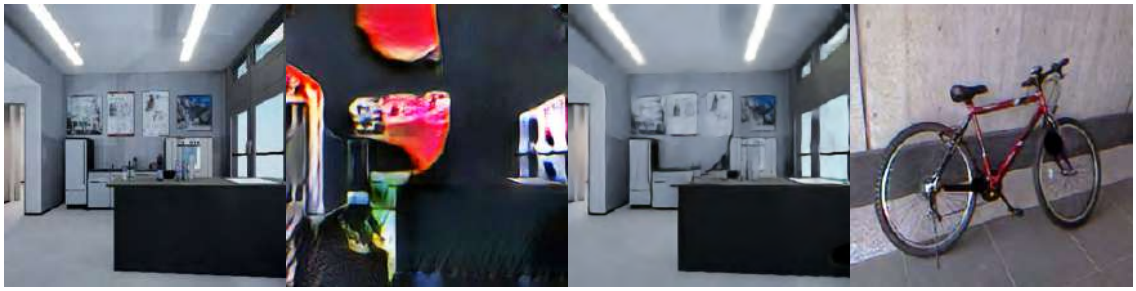


Abb. 8.21.: *real_B_164* ***Abb. 8.22.:*** *fake_A_164* ***Abb. 8.23.:*** *rec_B_164* ***Abb. 8.24.:*** *idt_B_164*

In diesem ersten Training fallen bereits einige Probleme mit den verwendeten Datensätzen auf:

- Auf sehr vielen der gefälschten Bilder, insbesondere aus früheren Generationen, lassen sich (mal mehr, mal weniger stark ausgeprägtes) Rastermuster oder ähnliche Artefakte erkennen (Abb. 8.2, Abb. 8.6). In diesem Experiment hat das Modell gelernt diese Strukturen zu vermeiden, daher tauchen sie in späteren Generationen nur noch äußerst selten auf.

- Besonders in späteren Generationen wurden bestimmte Muster aus der Domäne A erlernt, welche sich dann in den gefälschten Bildern wiederfinden (Abb. 8.10 (hier werden Teile eines Fahrrads erkannt)), Abb. 8.14 (hier wird die Lehne eines Bürostuhls

erkannt)). Diese Fehlinterpretationen von Objekten scheinen sich zu häufen, je länger die Netze trainiert wurden. Der Ursprung dieser Fehler liegt vermutlich an den relativ kleinen Datensätzen aus beiden Domänen.

- Bei vielen Bildern ist gar nicht ersichtlich, welche Strukturen das Modell zu erkennen glaubt. Die Resultate sind für Menschen schwer zu entziffern oder überhaupt nachzuvollziehen (Abb. 8.18, Abb. 8.22).

Es gibt allerdings auch bereits erwähnenswerte gute Resultate, zum Beispiel:

Abb. 8.25.: *real_B_122*

Abb. 8.26.: *fake_A_122*

Abb. 8.27.: *rec_B_122*

Abb. 8.28.: *idt_B_122*

Abb. 8.29.: *real_B_185*

Abb. 8.30.: *fake_A_185*

Abb. 8.31.: *rec_B_185*

Abb. 8.32.: *idt_B_185*

Abb. 8.33.: *real_B_186*

Abb. 8.34.: *fake_A_186*

Abb. 8.35.: *rec_B_186*

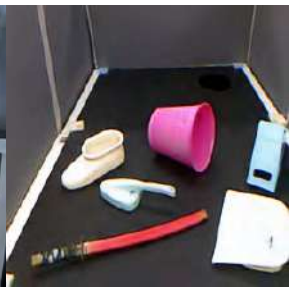


Abb. 8.36.: *idt_B_186*

8.2. unreal2kinect2

Da im der reale Datensatz im vorherigen Experiment vermutlich zu klein war und viele Objekte wiederholt auftauchten, wurde für dieses Experiment der vollständige Datensatz von ViDRILO[21] verwendet.

Der Datensatz mit virtuellen Szenen stammt wieder der RobCoG-Küche[22][23] und wurde vom Autor dieser Arbeit aufgenommen. Der Datensatz aus dem vorherigen Experiment wurde jedoch um Bilder mit Requisiten ergänzt um die zahlreichen rechteckigen Formen der Küche etwas aufzubrechen und so das Netz daran zu hindern, diese Formen zu erlernen. Zudem wurde eine neuere Version der Küche verwendet, welche unter anderem realistischeres Licht beinhaltet. Dieser Datensatz war etwa doppelt so groß wie im vorherigen Experiment.

Abb. 8.37.: *real_B_040*

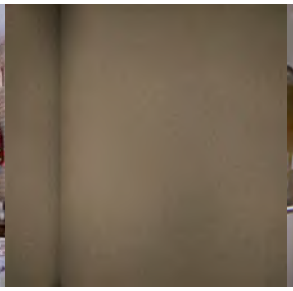


Abb. 8.38.: *fake_A_040*

Abb. 8.39.: *rec_B_040*

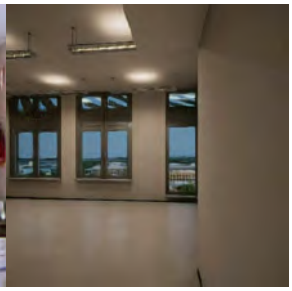


Abb. 8.40.: *idt_B_040*

Abb. 8.41.: *real_B_043*

Abb. 8.42.: *fake_A_043*

Abb. 8.43.: *rec_B_043*

Abb. 8.44.: *idt_B_043*

Abb. 8.45.: *real_B_036*

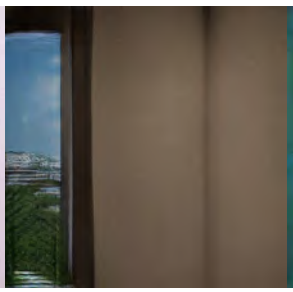


Abb. 8.46.: *fake_A_036*

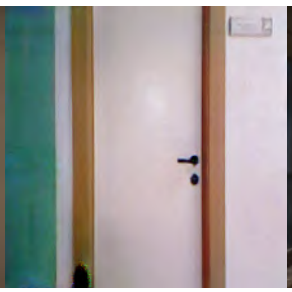


Abb. 8.47.: *rec_B_036*

Abb. 8.48.: *idt_B_036*

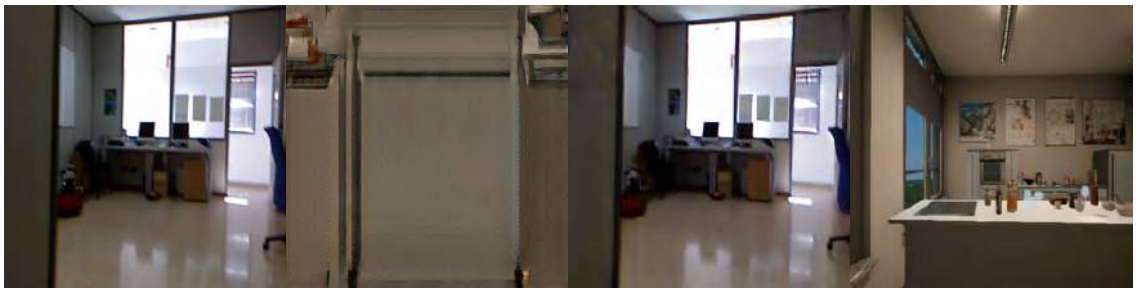


Abb. 8.49.: *real_B_050* ***Abb. 8.50.:*** *fake_A_050* ***Abb. 8.51.:*** *rec_B_050* ***Abb. 8.52.:*** *idt_B_050*

- ViDRILO enthält etwa 22000 Bilder. Dieser Trainingsvorgang musste nach einigen Tagen abgebrochen werden, da das Training mit dem vollständigen Datensatzt einige Wochen gedauert hätte, was den Rahmen dieser Arbeit gesprengt hätte. Da dieser Trainingsvorgang kürzer war als die anderen, gab es auch nur wenige „gute“ Ergebnisse. Es ist jedoch nicht auszuschließen, dass derartig umfangreiche Trainigsvorgänge gute Resultate liefern könnten.

- Ein auffälliges Problem bei diesem Experiment ist, dass sehr viele (nahezu) leere Bilder generiert wurden (Abb. 8.38, Abb. 8.42), insbesondere in späteren Generationen. Da der Datensatz an gerenderten Szenen einige „leere“ Bilder enthielt ist es für den Diskriminator nicht möglich, „leere“ generierte Bilder von „leeren“ echten Bildern zu unterscheiden. Durch mode collapse kann der Generator dann lernen, dass er mit „leeren“ Bildern den Diskriminator täuschen kann. In zukünftigen Trainingsvorgängen müssen diese „leeren“ Bilder also entfernt werden.

- Wie im vorherigen Trainingsvorgang auch hat das Netz einige Strukturen erlernt, welche dann auch in anderen Strukturen wiedererkannt werden. Zum Beispiel Fenster (Abb. 8.46) oder eine Schublade (Abb. 8.50). In zukünftigen Trainingsvorgängen wird versucht dem entgegenzuwirken, indem der Datensatz virtuell gerenderter Szenen größere Vielfalt aufweist (zum Beispiel durch Bilder, welche nicht aus der RobCoG-Küche stammen).

8.3. unreal2kinect3

Aufgrund der Probleme im vorherigen Trainingsvorgang wurde in diesem (und allen weiteren) die Größe der Datensätze bewusst kleiner gehalten. Für den realen Datensatz wurden knapp 1600 zufällig aus ViDRILO ausgewählte Bilder verwendet.

Statt Bildern aus der RobCoG-Küche wurden für dieses Experiment Bilder in von Unreal Engine 4 mitgeliefertem Beispielinhalten aufgenommen, um die Vielfältigkeit des virtuellen Datensatzes zu erhöhen. Hiermit wird verhindert, dass bestimmt Strukturen aus der RobCoG-Küche während des Trainings erlernt und wiedererkannt werden.

Abb. 8.53.: *real_B_172*

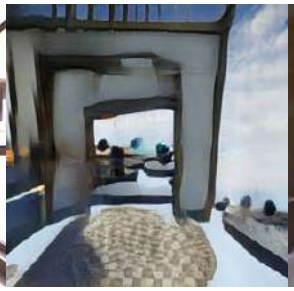


Abb. 8.54.: *fake_A_172*

Abb. 8.55.: *rec_B_172*

Abb. 8.56.: *idt_B_172*

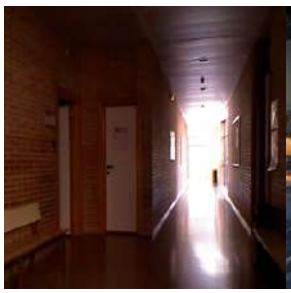


Abb. 8.57.: *real_B_146*

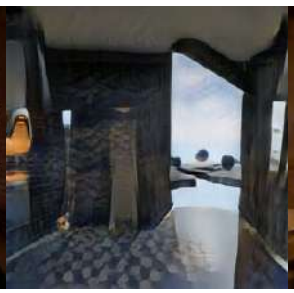


Abb. 8.58.: *fake_A_146*

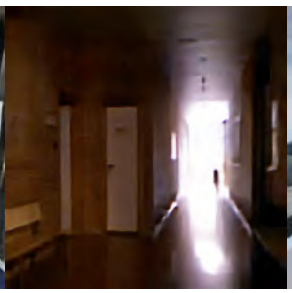


Abb. 8.59.: *rec_B_146*

Abb. 8.60.: *idt_B_146*

Abb. 8.61.: *real_B_094*

Abb. 8.62.: *fake_A_094*

Abb. 8.63.: *rec_B_094*

Abb. 8.64.: *idt_B_094*

Abb. 8.65.: *real_B_168* ***Abb. 8.66.:*** *fake_A_168* ***Abb. 8.67.:*** *rec_B_168* ***Abb. 8.68.:*** *idt_B_168*

- Trotz des größeren und vielfältigeren Datensatzes an realen Bildern besteht weiterhin ein Problem mit Overfitting, allerdings sind die in diesem Vorgang erkannten Strukturen für Menschen meist schwer nachvollziehbar. Sind die erkannten Strukturen auch für Menschen erkennbar, so handelt es sich meist um Fenster (Abb. 8.54) oder Flure (Abb. 8.58).

- Da der Datensatz von ViDRILO einige sehr dunkle Bilder beinhaltet, werden auch teils sehr dunkle Bilder generiert, selbst wenn das Ursprungsbild gut erkennbar und hell war (Abb. 8.62).

- Ähnlich den vorherigen Trainingsvorgängen sind auch in diesem in einigen generierten Bildern Rastermuster zu erkennen (Abb. 8.66). Je länger das Netzwerk trainiert wird und je größer die Input-Datensätze sind, desto seltener sollten solche Artefakte werden, sie sind allerdings kaum gänzlich vermeidbar.

8.4. unreal2kinect4 und unreal2kinect4_2

Für diesen Trainingsvorgang wurden die gleichen „echten“ Bilder wie im vorherigen Vorgang verwendet. Dabei stammen die Bilder mit der Bezeichnung *fake_A_1* vom gleichen Modell und die Bilder mit der Bezeichnung *fake_A_2* von einem anderen Modell.

Der zweite Datensatz ist eine Kombination der beiden vorhergegangenen virtuellen Datensätze. Dies ist wieder mit dem Ziel erfolgt, eine besonders hohe Vielfalt in den Bildern des virtuellen Datensatzes zu gewährleisten.

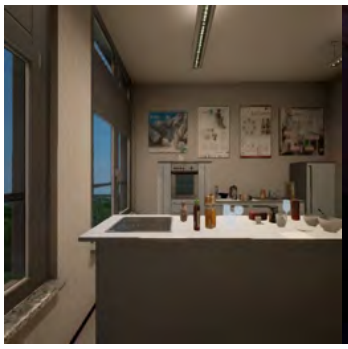


Abb. 8.69.: *real_B*

Abb. 8.70.: *fake_A_1_1*

Abb. 8.71.: *fake_A_2_1*

Abb. 8.72.: *real_B*

Abb. 8.73.: *fake_A_1_2*

Abb. 8.74.: *fake_A_2_2*

Abb. 8.75.: *real_B*

Abb. 8.76.: *fake_A_1_3*

Abb. 8.77.: *fake_A_2_3*

Es ist also zu beobachten, dass CycleGAN trotz exakt identischer Eingabeparameter und Inputdatensätzen, dennoch deutlich unterschiedliche Modelle trainieren kann. Dies könnte allerdings auch durch die Laufzeit von 200 Generationen („epochs“) bedingt sein. Die grundsätzliche Struktur der Eingabebilder bleibt jedoch bei beiden Modellen gut erhalten.

Nicht nachvollziehbar ist, warum in bestimmten Fällen besonders dunkle Bilder generiert werden (Abb. 8.69, 8.70, 8.71) und warum dies offenbar eine Gemeinsamkeit der ansonsten sichtbar unterschiedlichen Modelle darstellt.

8.5. unreal2kinect5

Da sowohl die Trainingsvorgänge mit Bildern aus der RobCoG-Küche als auch die Trainingsvorgänge mit Bildern aus Beispielinhalten (und die Kombination der beiden) einige Overfitting-Probleme hinsichtlich in den Datensätzen vorkommenden Strukturen hatten, wurde diesmal ein virtueller Datensatz aus einer dritten Quelle verwendet. Die Bilder für den hier verwendeten Datensatz wurden im Epic Zen Garden[3] aufgenommen.

Der reale Datensatz ist der gleiche, wie auch in den vorherigen Experimenten.

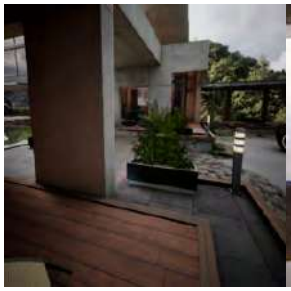


Abb. 8.78.: *real_B_200*

Abb. 8.79.: *fake_A_200*

Abb. 8.80.: *rec_B_200*

Abb. 8.81.: *idt_B_200*

Abb. 8.82.: *real_B_199*

Abb. 8.83.: *fake_A_199*

Abb. 8.84.: *rec_B_199*

Abb. 8.85.: *idt_B_199*

Abb. 8.86.: *real_B_197*

Abb. 8.87.: *fake_A_197*

Abb. 8.88.: *rec_B_197*

Abb. 8.89.: *idt_B_197*

[3] `https://www.unrealengine.com/marketplace/en-US/slug/epic-zen-garden`

Auffällig ist hier, dass, Bilder aus dem Epic Zen Garden nur sehr schlecht zu Kinect-artigen Bildern umgewandelt werden können, selbst in sehr späten Generationen des Trainings. Dies könnte dadurch zu erklären sein, dass die Kinect-Bilder aus dem Trainingsdatensatz mehr Ähnlichkeiten zu Bildern aus der RobCoG-Küche aufweisen als zu Bildern aus dem Epic Zen Garden.

Wird das trainierte Modell jedoch auf Bilder aus der RobCoG-Küche angewandt, so sind die Resultate deutlich besser. Es besteht große Übereinstimmung zwischen der Struktur von Eingabebildern und generierten Bildern.

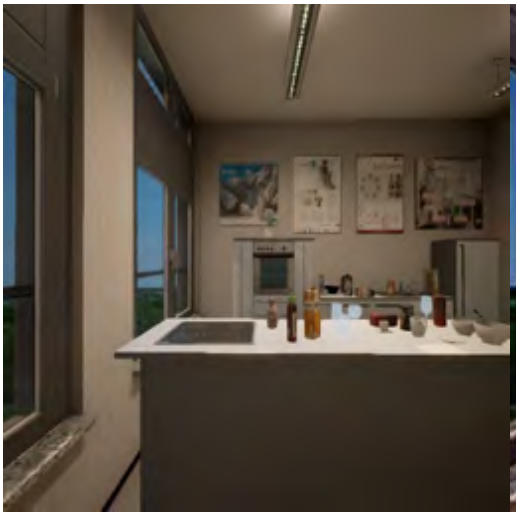


Abb. 8.90.: *real_B*

Abb. 8.91.: *fake_A*

8.6. unreal2kinect6

Für diesen Trainingsvorgang wurden schließlich die Datensätze aus der RobCoG-Küche, aus dem Beispielinhalt und aus dem Epic Zen Garden zu einem großen, vielfältigen Datensatz kombiniert. Der reale Datensatz blieb im Vergleich zu den vorherigen beiden Experimenten unverändert.

Abb. 8.92.: *real_B_196* ***Abb. 8.93.:*** *fake_A_196* ***Abb. 8.94.:*** *rec_B_196* ***Abb. 8.95.:*** *idt_B_196*

Abb. 8.96.: *real_B_194* ***Abb. 8.97.:*** *fake_A_194* ***Abb. 8.98.:*** *rec_B_194* ***Abb. 8.99.:*** *idt_B_194*

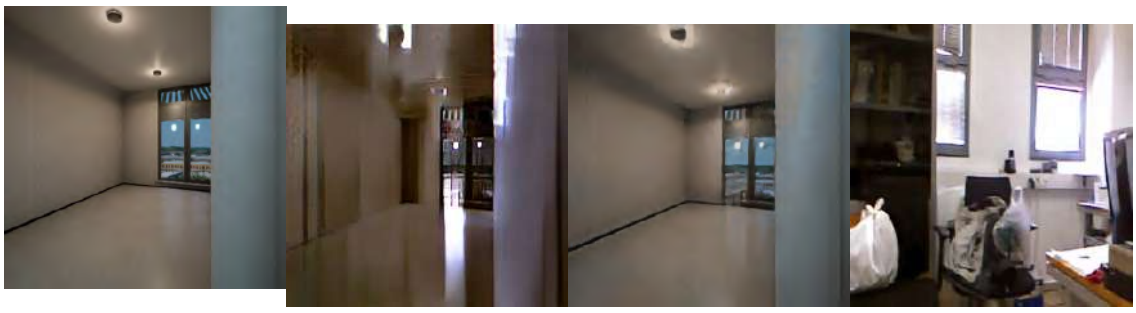


Abb. 8.100.: *real_B_187* ***Abb. 8.101.:*** *. . . 187* ***Abb. 8.102.:*** *rec_B_187* ***Abb. 8.103.:*** *idt_B_187*

Subjektiv, menschlich betrachtet ist auffällig, dass die Qualität der Resultate sich durch den größeren Datensatz nicht verbessert hat (insbesondere im Vergleich zu Abb. 8.90, 8.91). Dies

könnte dadurch zu begründen sein, dass die größere Menge Inputdaten auch einen längeren Trainingszeitraum gebraucht hätte.

Kapitel 9

Fazit

Ziel dieser Bachelorarbeit war es, RGB-Sensormodelle anhand virtueller Umgebungen zu erlernen. Zur Realisierung wurde versucht, ein Modell so zu trainieren, dass es aus gerenderten Szenen als Input möglichst realitätsnahe Bilder als Output generieren kann.

Früher wurden in der Bildverarbeitung meistens diskriminative Modelle verwendet, also solche die Inputdaten bestimmten Klassen zuordnen. In der vorliegenden Arbeit ging es jedoch nicht darum, Bilder zu klassifizieren, sondern es sollten neue Bilder generiert werden. Für diese Aufgabe bot sich somit eine generative Modellierung an. Da es um die Verarbeitung hochdimensionaler Bilddaten ging, wurden zum Erlernen der komplexen hierarchischen Strukturen tiefe neuronale Netze verwendet. Dazu wurden aus der Vielzahl von generativen Modellen GANs gewählt, da diese ausgehend von einem zufälligen Rauschen aus Beispielen eigenständig erlernen neue, so nicht in den Trainingsdaten vorkommende Bildern,zu generieren.

Die generierten Bilder sollten möglichst wenig von realen Bildern (also die Wahrnehmung der Realität durch einen Sensor) abweichen. Selbst der mit einem komplizierte Sehsystem ausgestattete Mensch sollte möglichst nicht mehr erkennen können, ob die Bilder generiert oder echt sind. Zur Überprüfung der Übereinstimmung von echten und generierten Bildern dient bei CycleGAN die KL-Divergenz.

Ein GAN-Training kann als Min-Max-Spiel verstanden werden, in dem die gegnerischen Netze das Nash Equilibrium anstreben. GANs suchen also kein bereits feststehendes Minimum der Optimierung, sondern es handelt sich um ein dynamisches, schwer trainierbares System. Dies zeigte sich auch bei den im Rahmen dieser Arbeit durchgeführten Experimenten. In den Expertimenten traten diverse Probleme wie Vanishing Gradient, Overfitting und Mode Collapse.

Zunächst wurde versucht, die Ergebnisse aus „Unpaired Image to Image Translation using Cycle-Consistent Adversarial Networks“[2] zu reproduzieren. Doch trotz gleicher Trainingsdatensätze und Parameter waren die Resultate nicht ideal: Es traten rasterartige Muster auf, die so im Originalbild nicht vorkamen. Außerdem kam es zu Overfitting.

Anschließend wurden verschiedene Modelle selbst trainiert. Für alle selbst trainierten Modelle wurde die gleiche Anzahl an Generationen verwendet, wodurch sich für die unterschiedlichen

Datenmengen zum Teil stark voneinander abweichende, kaum vorhersehbare Trainingszeiten ergaben. Verändert wurde bei den Experimenten die Herkunft, also Art und Vielfalt der Daten, sowie die Datenmenge, die trainiert wurde.

Abschließend ist festzuhalten, dass sowohl bei den reproduzierten als auch bei den eigenen Experimenten gute wie schlechte Ergebnisse bezüglich der Übereinstimmung von echten und generierten Bildern beobachtet werden konnten. Die besten Ergebnisse wurden bei recht großer Datenvielfalt und mit gerade so großen Datenmengen erzielt, das diese scheinbar im vorgegebenen Trainingszeitraum verarbeitet werden konnten. Es ließ sich aus zeitlichen wie praktischen Gründen nicht weiter klären, welchen genauen Einfluss die verschiedenen Parameter haben, denn die Datenvielfalt und -menge lässt sich nicht auf die Anzahl und Größe der Bilder reduzieren, sondern ist auch abhängig von den Inhalten. Dies führt zu kaum vorhersehbaren Trainingszeiten.

Kapitel 10

Ausblick

Es ist zu vermuten, dass Bildtransformationen durch maschinelles Lernen, wie sie in dieser Arbeit angestrebt wurden, möglich sind. Zwar liefert CycleGAN teils sinnvolle Ansätze für die zentrale Fragestellung dieser Arbeit, ob dieser Algorithmus jedoch die beste Möglichkeit darstellt, das Problem anzugehen, bleibt zweifelhaft. Sicher ist jedoch, dass um sinnvollere Ergebnisse zu erhalten, umfangreichere und besser kontrollierte Lernvorgänge notwendig sind und dabei auch verschiedene andere Algorithmen untersucht werden sollten.

Zu überprüfen wäre auch, warum zwei Instanzen von CycleGAN mit gleichen Inputdaten und Eingabeparametern immer zu unterschiedlichen Ergebnissen kommen. Auch sollte der Frage nachgegangen werden, ob die Ergebnisse bei längeren Trainingsperioden irgendwann konvergieren, denn derartig lange Trainingsvorgänge waren im Rahmen dieser Arbeit nicht realisierbar. Interessant wäre auch nachzuforschen, warum genau teilweise dunkle Bilder generiert werden, selbst wenn die Trainingsdatensätze frei von derartigen Bildern sind.

Es bleibt fragwürdig, ob CycleGAN überhaupt für die Anwendung im Umwandeln von virtuell generierten Bildern zu möglichst real wirkenden Bildern geeignet ist, oder ob ein herkömmlicher Filter nicht bessere Resultate liefern würde.

Literaturverzeichnis

[1] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets, 2014.

[2] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Computer Vision (ICCV), 2017 IEEE International Conference on*, 2017.

[3] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. *CoRR*, abs/1312.6114, 2014.

[4] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, pages II–1278–II–1286. JMLR.org, 2014.

[5] Aäron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks. *CoRR*, abs/1601.06759, 2016.

[6] Aäron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu. Conditional image generation with pixelcnn decoders. *CoRR*, abs/1606.05328, 2016.

[7] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[8] Felix A. Gers, Jürgen A. Schmidhuber, and Fred A. Cummins. Learning to forget: Continual prediction with lstm. *Neural Comput.*, 12(10):2451–2471, October 2000.

[9] Jason Cong and Bingjun Xiao. Minimizing computation in convolutional neural networks. In *ICANN*, 2014.

[10] Francois Chollet. *Deep Learning with Python*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2017.

[11] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. O'Reilly Media, Inc., 2017.

[12] Josh Patterson and Adam Gibson. *Deep Learning A Practitioner's Approach*. O'Reilly Media, Inc., 2017.

[13] H. Bourlard and Y. Kamp. Auto-association by multilayer perceptrons and singular value decomposition. *Biol. Cybern.*, 59(4-5):291–294, September 1988.

[14] Geoffrey E. Hinton and Richard S. Zemel. Autoencoders, minimum description length and helmholtz free energy. In *Proceedings of the 6th International Conference on Neural Information Processing Systems*, NIPS'93, pages 3–10, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[15] Douwe Ousinga. *Deep Learning Cookbook*. O'Reilly Media, Inc., 2018.

[16] Geoffrey E Hinton and James L. McClelland. Learning representations by recirculation. In D. Z. Anderson, editor, *Neural Information Processing Systems*, pages 358–366. American Institute of Physics, 1988.

[17] G E Hinton and R R Salakhutdinov. Reducing the dimensionality of data with neural networks. 313, 2006.

[18] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 1096–1103, New York, NY, USA, 2008. ACM.

[19] Anders Boesen Lindbo Larsen, Søren Kaae Sønderby, Hugo Larochelle, and Ole Winther. Autoencoding beyond pixels using a learned similarity metric. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pages 1558–1566. JMLR.org, 2016.

[20] Quanshi Zhang, Xuan Song, Xiaowei Shao, Huijing Zhao, and Ryosuke Shibasaki. Category modeling from just a single labeling: Use depth information to guide the learning of 2d models. In *IEEE International Conference on Computer Vision and Pattern Rcognition (CVPR)*, 2013.

[21] J. Martinez-Gomez, M. Cazorla, I. Garcia-Varea, and V. Morell. Vidrilo: The visual and depth robot indoor localization with objects information dataset. In *International Journal of Robotics Research*, 2013.

[22] Andrei Haidu and Michael Beetz. Automated models of human everyday activity based on game and virtual reality technology. In *International Conference on Robotics and Automation (ICRA)*, 2019. Accepted for publication.

[23] Andrei Haidu, Daniel Bessler, Asil Kaan Bozcuoglu, and Michael Beetz. Knowrob_sim - game engine-enabled knowledge processing towards cognition-enabled robot control. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2018, Madrid, Spain, October 1-5, 2018*, 2018.

Anhang A

Anhang

Abb. A.1.: *Inhalte der realen Datensätze der jeweiligen Trainingsdurchläufe*

Trainingsdurchlauf	Quanshi Zhang[20]	ViDRILO[21]
unreal2kinect	Ja	Nein
unreal2kinect2	Nein	Ja, vollständig
unreal2kinect3	Nein	Ja, Teilmenge
unreal2kinect4	Ja	Ja, Teilmenge
unreal2kinect5	Ja	Ja, Teilmenge
unreal2kinect6	Ja	Ja, Teilmenge

Abb. A.2.: *Inhalte der virtuellen Datensätze der jeweiligen Trainingsdurchläufe*

Trainingsdurchlauf	RobCoG-Küche	Unreal Sample Content	Epic Zen Garden
unreal2kinect	Ja (ohne Props)	Nein	Nein
unreal2kinect2	Ja (mit Props)	Nein	Nein
unreal2kinect3	Nein	Ja	Nein
unreal2kinect4	Ja (mit Props)	Ja	Nein
unreal2kinect5	Nein	Nein	Ja
unreal2kinect6	Ja (mit Props)	Ja	Ja