

Generierung künstlicher Trainingsdaten für die Straßenschilderkennung in Fahrzeugen mittels Generative Adversarial Networks

Studienarbeit

Studiengang Informatik

an der Dualen Hochschule Baden-Württemberg Stuttgart

von

Frederik Esau

08.06.2023

Bearbeitungszeitraum
Matrikelnummer, Kurs
Betreuer

24.10.2022 - 08.06.2023
6526552, TINF20ITA
Prof. Dr. Monika Kochanowski

Erklärung

Ich versichere hiermit, dass ich meine Studienarbeit mit dem Thema: *Generierung künstlicher Trainingsdaten für die Straßenschilderkennung in Fahrzeugen mittels Generative Adversarial Networks* selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Stuttgart, 08.06.2023

Frederik Esau

Abstract

Abstract normalerweise auf Englisch. Siehe: http://www.dhbw.de/fileadmin/user/public/Dokumente/Portal/Richtlinien_Praxismodule_Studien_und_Bachelorarbeiten_JG2011ff.pdf (8.3.1 Inhaltsverzeichnis)

Ein „Abstract“ ist eine prägnante Inhaltsangabe, ein Abriss ohne Interpretation und Wertung einer wissenschaftlichen Arbeit. In DIN 1426 wird das (oder auch der) Abstract als Kurzreferat zur Inhaltsangabe beschrieben.

Objektivität soll sich jeder persönlichen Wertung enthalten

Kürze soll so kurz wie möglich sein

Genauigkeit soll genau die Inhalte und die Meinung der Originalarbeit wiedergeben

Üblicherweise müssen wissenschaftliche Artikel einen Abstract enthalten, typischerweise von 100-150 Wörtern, ohne Bilder und Literaturzitate und in einem Absatz.

Quelle: <http://de.wikipedia.org/wiki/Abstract> Abgerufen 07.07.2011

Diese etwa einseitige Zusammenfassung soll es dem Leser ermöglichen, Inhalt der Arbeit und Vorgehensweise des Autors rasch zu überblicken. Gegenstand des Abstract sind insbesondere

- Problemstellung der Arbeit,
- im Rahmen der Arbeit geprüfte Hypothesen bzw. beantwortete Fragen,
- der Analyse zugrunde liegende Methode,
- wesentliche, im Rahmen der Arbeit gewonnene Erkenntnisse,
- Einschränkungen des Gültigkeitsbereichs (der Erkenntnisse) sowie nicht beantwortete Fragen.

Quelle: http://www.ib.dhbw-mannheim.de/fileadmin/ms/bwl-ib/Downloads_alt/Leitfaden_31.05.pdf, S. 49

Inhaltsverzeichnis

Abkürzungsverzeichnis	V
Abbildungsverzeichnis	VI
Tabellenverzeichnis	VII
Listings	VIII
1 Einleitung	1
1.1 Problemstellung	1
1.2 Vorgehensweise	1
1.3 Ziel der Arbeit	1
2 Stand der Technik	2
2.1 Straßenschilderkennung	2
2.2 Künstliche Neuronale Netze	4
2.2.1 Training	6
2.2.2 Convolutional Neural Networks	9
2.3 Bildgenerierung mittels Künstlicher Neuronaler Netze	10
2.3.1 Mathematischer Hintergrund	11
2.3.2 Pixel Recurrent Neural Networks	12
2.3.3 Autoencoder	15
2.3.4 Generative Adversarial Networks	17
2.4 Vorherige Arbeiten	22
2.4.1 Generierung Taiwanischer Straßenschilder mittels DCGAN	23
2.4.2 Generierung Deutscher Straßenschilder mittels CycleGAN	24
2.5 Machine Learning Frameworks	25
3 Konzeption des Modells	27
3.1 Datensatz	27
3.2 Framework	30
3.3 Architektur	32
3.4 Datenaugmentation	33
3.5 Training	34
4 Implementierung und Training	36
4.1 Modell	36
4.1.1 Konstruktor	37
4.1.2 fit-Methode	39
4.1.3 generate-Methode	41

4.2	Datenaugmentierung	41
4.2.1	Skalierung	42
4.2.2	Rotation	43
4.3	Training	44
4.3.1	Laden der Datensätze	44
4.3.2	Ausführen des Trainings	46
4.3.3	Logging	46
4.3.4	Vorgehensweise	46
4.4	Trainingsergebnisse	46
4.4.1	U-Net	47
4.4.2	ResNet	48
4.5	Generierung	49
5	Augmentation der generierten Bilder	51
5.1	Ungültige Straßenschilder	51
5.2	Bewegungsunschärfe	51
5.3	Schnee	52
6	Evaluation	53
6.1	Evaluation der Generierung	53
6.2	Evaluation der Augmentierung	53
6.3	Verbesserungsmöglichkeiten	53
7	Zusammenfassung	54
Anhang		59

Abkürzungsverzeichnis

CNN	Convolutional Neural Network
cGAN	Conditional Generative Adversarial Network
CycleGAN	Cycle-Consistent Generative Adversarial Network
GAN	Generative Adversarial Network
GPU	Grafikkarte (engl.: Graphics Processing Unit)
GTSRB	German Traffic Sign Recognition Benchmark
KNN	Künstliches Neuronales Netz
PixelRNN	Pixel Recurrent Neural Network
ResNet	Residual Neural Network
SSIM	Index struktureller Ähnlichkeit (engl.: Structural Similarity Index)
SVM	Support Vector Machine
TOML	Tom's Obvious Minimal Language

Abbildungsverzeichnis

2.1	Erschwerende Einflüsse auf die Straßenschilderkennung [5]	3
2.2	Weitere mögliche Einflüsse auf die Straßenschilderkennung	3
2.3	Einzelnes Neuron eines Künstliches Neuronales Netzs (KNNs) [9]	5
2.4	Vollständiges KNN (<i>angelehnt an [9]</i>)	5
2.5	Beispiel für eine Faltung (engl.: Convolution) [11]	10
2.6	Beispielhafter Vergleich von $\hat{p}(x)$ und $p(x)$ [13]	11
2.7	Taxonomie generativer Modelle [14]	12
2.8	Bestimmung von $\hat{p}(x)$ mit PixelRNNs [15]	13
2.9	Darstellung eines Recurrent Neural Networks [16]	14
2.10	Architektur eines Autoencoders	15
2.11	Zusammenspiel zwischen Generator und Diskriminatot	17
2.12	Beispielergebnisse der Generierung taiwanischer SChilder [27]	23
3.1	Beispielbilder aus dem GTSRB Datensatz [5]	27
3.2	Häufigkeitsverteilung der Klassen von Straßenschildern im präparierten German Traffic Sign Recognition Benchmark (GTSRB)	28
3.3	Beispielbilder aus der chinesischen Traffic Sign Recognition Database [32]	29
3.4	Beispielbild aus dem <i>Mapillary</i> Datensatz [34]	29
3.5	Häufigkeitsverteilung der Kategorien von Straßenschildern im präparierten Datensatz	30
3.6	Domänen für die Bild-zu-Bild Übersetzung	33
3.7	Rotation der Straßenschilder mittels eulerscher Winkel	34
3.8	Trainingsschritte des Cycle-Consistent Generative Adversarial Network (CycleGAN)	35
4.1	Modal Collaps des ResNet nach 200 Trainingsepochen	48
4.2	Positiv herausstechende Bilder des ResNets verschiedener Epochen	48
4.3	Negativ herausstechende Bilder des ResNets verschiedener Epochen	49
4.4	Beispielbilder des ResNet-basierten Modells mit 6 Residual Blocks	49

Tabellenverzeichnis

2.1	Vergleich der Objekterkennung mit und ohne künstliche Trainingsdaten	24
2.2	Vergleich der Klassifikation mit echten und künstlichen Trainingsdaten [28] . .	25
4.1	Auswahl an Methoden aus der CycleGAN Klasse	37
4.2	Vergleich von UNet und ResNet	47
4.3	Auswahl an Methoden aus der CycleGAN Klasse	50
6.1	Ergebnisse des Trainings eines VGG16 Klassifikators	53

Listings

4.1	model.py - Auswahl der Generator-Architektur	38
4.2	model.py - Initialisierung der Generatoren	39
4.3	model.py - <code>fit</code> -Methode	40
4.4	model.py - <code>generate</code> -Methode	41
4.5	Skalieren der Bild-Tensoren	42
4.6	Augmentierung eines Batches von Bildern	43
4.7	train.py - Laden des Trainingsdatensatzes	45
4.8	train.py - Laden des Trainingsdatensatzes	46
5.1	Hinzufügen von Schnee: Funktionsdeklaration	52
1	Augmentierung eines Batches von Bildern	64
2	Augmentierung eines Batches von Bildern	66

1 | Einleitung

Während in großen Teilen des letzten Jahrhunderts Innovationen in der Fahrzeugentwicklung vor allem im Bereich der mechanischen Leistung und Effizienz stattgefunden haben, erwartet man zukünftige Verbesserungen im Automobil besonders softwareseitig [1]. ...

1.1 Problemstellung

1.2 Vorgehensweise

1.3 Ziel der Arbeit

2 | Stand der Technik

2.1 Straßenschilderkennung

Eine Straßenschilderkennung zählt mittlerweile zu der Standardausstattung vieler Neuwagen. Im Jahr 2024 tritt zudem eine EU-Verordnung in Kraft, durch die sämtliche neu produzierten Fahrzeuge mit einer solchen Funktion ausgestattet werden müssen [2]. Daran zeigt sich, dass das Thema bereits weitreichend etabliert ist.

Straßenschilder werden zu folgendem Zweck eingesetzt: Es sollen Informationen über die Verkehrssituation und über geltende Vorschriften des Gebiets, in dem sich das Fahrzeug zu einem gegebenen Zeitpunkt befindet, präsentiert werden. Durch Straßenschilder werden unter anderem Geschwindigkeitsvorgaben, Gefahrenhinweise und allgemeine Verkehrsregeln kommuniziert. Dabei sind die Schilder so konzipiert, dass sie sich visuell möglichst von ihrem Hintergrund abheben und leicht voneinander zu unterscheiden sind. Automatische Straßenschilderkennungen können Fahrer*innen in Situationen unterstützen, in denen sie Schilder übersehen oder gezielt missachten. Anstelle dass ein reales Schild beispielsweise nur für einige Sekunden sichtbar ist, bevor es außerhalb der Sichtweite des Fahrzeugführenden ist, ist eine durchgehende Anzeige auf den Displays eines Fahrzeugs möglich. Auch akustische Warnungen oder ein aktives Eingreifen von Sicherheitssystemen sind denkbar, beziehungsweise bereits in Serienfahrzeugen vorhanden.

Eine Straßenschilderkennung erfolgt visuell und wird somit mittels Kameras umgesetzt. Dabei lassen sich viele Schilder durch eine bestimmte Form (Kreis, Dreieck, Achteck, etc.) und ein Piktogramm (Schneeflocke, Person, etc.) oder eine Zahl identifizieren. Somit können die verschiedenen Arten von Straßenschilder in Klassen unterteilt werden, die durch den Erkennungsalgorithmus detektiert werden. Für die praktische Umsetzung solcher Algorithmen werden häufig KNNs verwendet. Diese werden in Kapitel 2.2 thematisch aufgeführt. Besonders relevant für das Thema dieser Studienarbeit ist, wie zuverlässig die momentan ausgelieferten Straßenschilderkennungen sind und welche Situationen die Algorithmen am ehesten zu falschen Aussagen verleiten. Auf dieser Grundlagen kann sich orientiert werden, welche Arten von Bildern vermehrt generiert werden sollen, um die Straßenschilderkennung verbessern zu können.

Im Jahr 2019 hat eine Automobilzeitschrift die Straßenschilderkennung von unterschiedlichen Fahrzeugherstellern getestet [3]. Zudem existieren verschiedene Publikationen, die sich mit

der Thematik befassen [4]. Die Ergebnisse des Zeitschriftenartikels weisen darauf hin, dass die Straßenschilderkennung einiger Fahrzeuge bereits weitreichend funktioniert. Geschwindigkeitsvorgaben werden überwiegend erkannt und dem Fahrer auf einem Display angezeigt. Auch existieren bereits akustische Warnungen bei einer Überschreitung der Höchstgeschwindigkeit. Dennoch existieren einige Situationen, die bei mehreren Fahrzeugen zu Problemen bei der Straßenschilderkennung geführt haben. Einen exemplarischen Überblick soll die folgende Grafik bieten: [3]



Abbildung 2.1: Erschwerende Einflüsse auf die Straßenschilderkennung [5]

Fahrzeuge verschiedener Hersteller haben in den Tests Aufhebungsschilder nicht korrekt interpretiert. Damit sind Schilder gemeint, die entweder Geschwindigkeitsbegrenzungen oder Überholverbote außer Kraft setzen. Des Weiteren sorgten mittels Klebestreifen als ungültig erklärte Schilder, in einigen Fällen Dunkelheit, beispielsweise in Tunneln, und sogenannte LED Wechselverkehrszeichen für falsche Aussagen. Auch wird teilweise nicht erkannt, wenn Schilder für eine kreuzende Straße gelten, statt für die Straße, auf der sich das Fahrzeug zu dem gegebenen Zeitpunkt befinden. Weitere Aspekte, die in dem Artikel nicht explizit genannt sind, aber laut einer Publikation von 2014 in der Vergangenheit zu Schwierigkeiten geführt haben, sind mitunter die folgenden: [4]

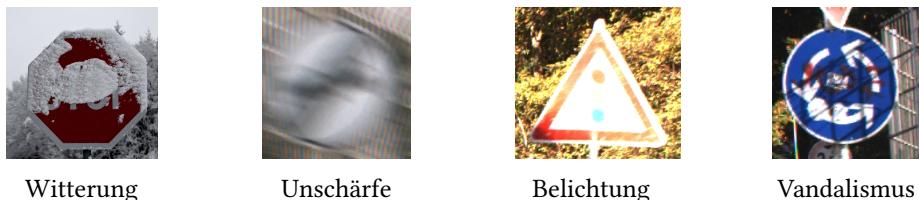


Abbildung 2.2: Weitere mögliche Einflüsse auf die Straßenschilderkennung

Eine weitere Publikation aus dem Jahre 2019 zeigt, dass die Qualität der Straßenschilderkennung von der Stärke der äußeren Einflüsse abhängt. Vergleichsweise geringfügig verdeckte Schilder hat das Testfahrzeug hier beispielsweise in der Regel korrekt klassifizieren können. Sobald eine größere Fläche des Schild verdeckt ist oder das Schild vermehrt beschmutzt ist, ist keine korrekte Klassifizierung erfolgt. Auch hier schreiben die Autoren, dass das Wetter und somit Sichtverhältnisse einen negativen Einfluss auf die Qualität der Erkennung zeigen. [6]

Die Erkenntnisse der Tests aus den genannten Artikeln geben Hinweise auf das folgende: In einigen genannten Situationen können sich die Fahrzeugführenden nicht vollständig auf die

Straßenschilderkennung ihrer Fahrzeuge verlassen. Ein Ziel der Hersteller ist das Anbieten von Fahrzeugen, die vollständig autonom, das heißt ohne menschliches Eingreifen, fahren können. Damit das möglich ist, muss die Software der Fahrzeuge auch solche Grenzfälle korrekt interpretieren. Das erfordert eine gewisse Menge an Daten, durch die diese Algorithmen trainiert werden.

Ziel dieser Arbeit ist ausgehend davon, gezielt Trainingsbilder erzeugen zu können, die einige dieser Aspekte simulieren. Es soll als alternative Möglichkeit dazu vorgeschlagen werden, sämtliche Trainingsdaten für Grenzfälle eigenständig in realen Fahrsituationen aufzunehmen.

2.2 Künstliche Neuronale Netze

Durch künstliche neuronale Netze (KNNs) können Maschinen lernen, bestimmte Probleme zu lösen, ohne dass ein Mensch vorher explizite Regeln dafür definieren muss. Dies steht im Kontrast zur Methode, der Maschine vorher einen festen, vollständigen Regelsatz bereitzustellen. Letztgenannter Ansatz zeigt in einigen Gebieten nur begrenzten Erfolg, da es für Menschen herausfordernd sein kann, Regelsätze für Vorgänge zu definieren, die unbewusst im Gehirn stattfinden oder viel Kontext erfordern. Zu nennen sind hierbei die visuelle Objekterkennung oder menschliche Sprache. Außerdem können neue, nicht in den Regeln beachtete Situationen dazu führen, dass die Maschine das Problem nicht mehr lösen kann. Die Grundidee hinter KNNs ist deshalb, dass sich die Maschine selber einen Wissensschatz aufbaut, der ihr beim Lösen des Problems hilft. Dies geschieht, indem die Entwickler ihr reale Trainingsbeispiele zeigen. Möchte man einen Algorithmus trainieren, der Schach spielen soll, kann man ihm beispielsweise eine Vielzahl an realen Schachpartien zeigen. Anhand dessen lernt der Algorithmus verschiedene Strategien und baut ein Spielverständnis auf, das womöglich über die menschlichen Fähigkeiten hinausgeht. [7, S. 1ff.]

In den letzten Jahrzehnten erlebte das maschinelle Lernen und damit auch das Gebiet der KNNs einen Aufschwung. Es existiert bereits seit Mitte des vergangenen Jarhunderts, wird allerdings erst durch die zunehmende Rechenleistung und die Verfügbarkeit von großen Datenmengen flächendeckend eingesetzt. Einsatzgebiete für KNNs sind unter anderem die Objekterkennung, das Verstehen von natürlicher Sprache und die Generierung von Text und Bildern. [8, S. 4, 17]

Die Inspiration für KNNs bildet die Informationsverarbeitung des Gehirns in Lebewesen. Die kleinste hier betrachtete Einheit ist das Neuron. Neuronen in KNNs sind konzeptionell inspiriert von realen, biologischen Neuronen, besitzen aber eine deutlich abstrahierte Funktionsweise. In KNNs berechnen sie ein Skalarprodukt ihrer gewichteten Eingangswerte, addieren einen sogenannten *Bias* hinzu und wenden auf das Ergebnis eine nichtlineare Funktion an. Letzere

wird auch als Aktivierungsfunktion bezeichnet. Diese Aktivierungsfunktion kann analog dazu gesehen werden, dass biologische Neuronen einen Schwellenwert (*engl.: threshold*) besitzen, der überschritten werden muss, damit das Neuron *feuert*, also einen Impuls an weitere Neuronen weitergibt. Aktivierungsfunktionen sind notwendig, damit neuronale Netze Probleme lösen können, die über die Fähigkeiten einer linearen Regression hinausgehen. Es wird eine nichtlineare Abhängigkeit zwischen dem Eingang X und dem Ausgang Y umgesetzt. [9]

In der Nachfolgenden Abbildung ist ein einzelnes künstliches Neuron eines KNNs darstellt. Das *Plus-Zeichen* steht für die Berechnung des Skalarprodukts der Eingänge und der darauf addierte Bias. Die Aktivierungsfunktion wird durch das *Sigmoid-Zeichen* symbolisiert. Der Ausgang (*rechts*) ist das Ergebnis der Berechnung des Neurons. [9]

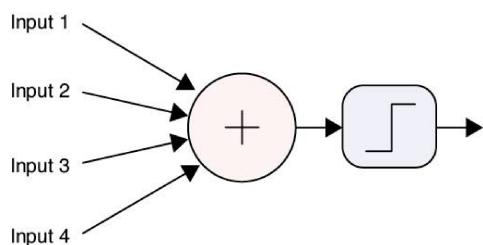


Abbildung 2.3: Einzelnes Neuron eines KNNs [9]

Um komplexe Probleme lösen zu können, werden mehrere Neuronen miteinander verbunden und in Schichten angeordnet. Jede Schicht erhält die Ausgangswerte der vorherigen Schicht als Eingang und gibt die daraus berechneten, neuen Werte an die nächste Schicht weiter. Abbildung 2.4 zeigt ein vollständiges KNN. Die Neuronen sind durch Kreise dargestellt und ihre Verbindungen durch Linien. [8]

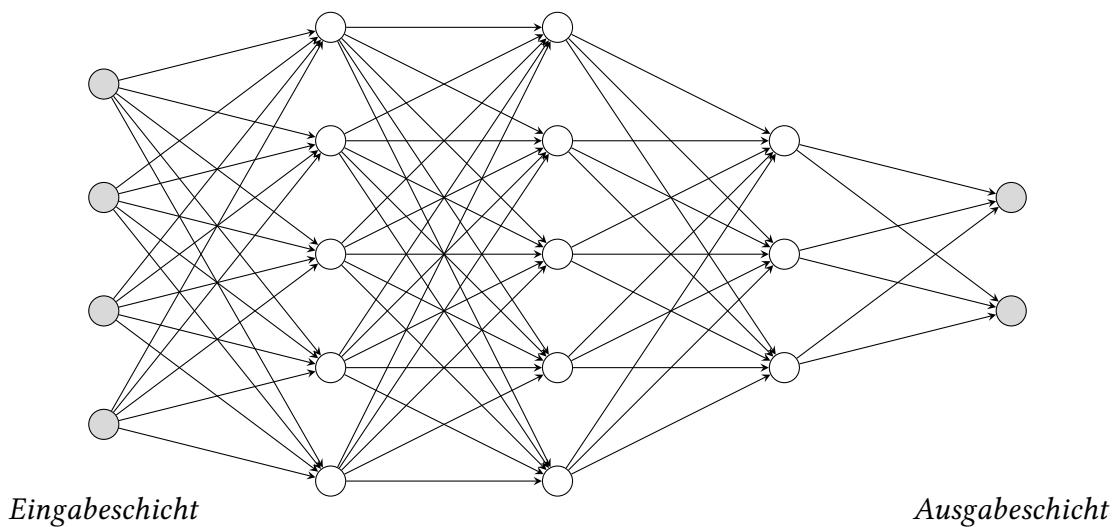


Abbildung 2.4: Vollständiges KNN (angelehnt an [9])

Eine Verbindung stellt dar, dass ein Neuron seinen berechneten Wert an das nachfolgende Neuron weitergibt. Dies geschieht hierbei ausschließlich von *links nach rechts*, womit das KNN als *Feedforward-Netzwerk* bezeichnet wird. Die Eingabeschicht erhält die Eingabewerte des Netzwerks, die Ausgabeschicht liefert die Vorhersage des Modells. Zwischen diesen beiden Schichten befinden sich beliebig viele verarbeitende Schichten, die als *Hidden Layer* bezeichnet werden. [8]

Die Vorhersage, gekennzeichnet durch die Werte der Ausgabeschicht, hängt von den jeweiligen Parametern der Neuronen des Netzwerks ab. Dies sind die Gewichte (*engl.: weights*) der Verbindungen zwischen den Neuronen sowie der Bias der Neuronen. Es existieren auch trainierbare Aktivierungsfunktionen, diese sind jedoch vergleichsweise unüblich. Entwickler sind für den Entwurf der Netzwerkarchitektur zuständig, die Parameter werden jedoch durch das Modell trainiert. Zu Beginn besitzt das Modell zufällige Parameter, wodurch es in der Regel nicht die gewünschte Abbildung zwischen Ein- und Ausgabe implementiert. Ein untrainierter Schachalgorithmus spielt demnach augenscheinlich willkürliche Züge. Ein untrainierter Katzenklassifikator besitzt keinen erkennbaren Wissensschatz darüber, welche Charakteristiken eine Katze optisch auszeichnen. Das Ziel des Trainings ist, dass die Parameter des Modells zunehmend gegen das Optimum konvergieren und so das Modell immer plausibler in seinen Vorhersagen wird. [8] [9]

2.2.1 Training

Für das Training von KNNs werden Daten benötigt. Zum einen ein Satz an *Trainingsdaten* und zum anderen ein Satz an *Testdaten*. Die Trainingsdaten dienen dazu, das Modell zu verbessern. Eine *Kostenfunktion* berechnet, wie genau die Vorhersagen des Modells auf den Trainingsdaten sind. Darauf basierend wird das Modell optimiert. Die Testdaten dienen zur Messung der Qualität des Modells. Es kann nämlich vorkommen, dass das KNN die Trainingsdaten *auswendig* lernt und deshalb hier gute Ergebnisse erzielt, aber eine unzureichende Performanz auf den Testdaten zeigt. Etwa wenn das KNN unerwartete Eigenschaften in den Trainingsdaten lernt. Aus diesem Grund werden trainierte Modelle nur anhand der Testdaten evaluiert. [8]

Die Kostenfunktion bestimmt die durchschnittliche Fehlerrate des KNNs auf einem gegebenen Datensatz. Sie berechnet somit, gemittelt über alle m Beispiele aus dem Datensatz, die Abweichung des vorhergesagten Wertes \hat{y} von dem tatsächlichen Wert y . Für ein einzelnes Beispiel aus dem Datensatz nutzt die Funktion dafür eine *Verlustfunktion* \mathcal{L} . Die Verlustfunktion bewertet somit eine einzelne Aussage des KNN, während die Kostenfunktion die durchschnitt-

liche Qualität der Aussagen auf dem gesamten Datensatz misst. Folgende Gleichung zeigt die Kostenfunktion: [7]

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) \quad (2.1)$$

Wobei:

J : Wert der Kostenfunktion

θ : Trainierbare Parameter des Modells

m : Anzahl der Trainingsbeispiele

i : Index des momentan betrachteten Trainingsbeispiels

\mathcal{L} : Verlustfunktion

\hat{y} : Vorhersage des Modells

y : Erwarteter Wert

Die Kostenfunktion J ist abhängig von einem θ . Das θ ist ein Vektor, der alle Gewichte und Biases und damit alle trainierbaren Parameter des KNNs beinhaltet. Verändern sich die Parameter des KNNs, dann liefert es andere Aussagen für das \hat{y} . Somit ändert sich der Wert der Kostenfunktion, wenn das KNN seine Parameter anpasst. [7]

Die Gleichung 2.1 kann auch mit dem Operator \mathbb{E} formuliert werden. Er beschreibt den Erwartungswert: [7]

$$J(\theta) = \mathbb{E}_{x,y}[\mathcal{L}(f(x; \theta), y)] \quad (2.2)$$

Was in Gleichung 2.1 das arithmetische Mittel der Verlustfunktionen ist, wird hier als Erwartungswert der Verlustfunktion geschrieben. Die Fragestellung lautet: *Wenn ein zufälliges Beispiel x und das zugehörige y aus dem Datensatz gezogen werden, was ist dann der erwartete Verlust des Modells?* Die Vorhersage \hat{y} des KNN wird hier als $f(x; \theta)$ geschrieben. Dabei ist f das KNN, das für ein gezogenes x eine bestimmte Vorhersage trifft. Diese Vorhersage ist zudem abhängig von θ . Diese alternative Darstellung der Kostenfunktion mit dem Operator \mathbb{E} ist relevant für Kapitel 2.3. [7]

Kostenfunktionen können verschiedene Arten von Verlustfunktionen verwenden. Für diese Arbeit sind sowohl die \mathcal{L}_1 und \mathcal{L}_2 Verlustfunktionen als auch der *Binary Crossentropy Loss* von besonderer Relevanz.

\mathcal{L}_1 Verlustfunktion Die \mathcal{L}_1 Verlustfunktion berechnet für ein gegebenes Beispiel die absolute Abweichung der Vorhersage von dem erwarteten Wert. Sie wird auch als *mittlere absolute Abweichung* (engl.: *mean absolute error*) bezeichnet.

$$\mathcal{L}_1 = |\hat{y} - y| \quad (2.3)$$

\mathcal{L}_2 Verlustfunktion Die \mathcal{L}_2 Verlustfunktion berechnet hingegen die quadratische Abweichung von dem erwarteten Wert. Hier haben somit große Abweichungen einen stärkeren Einfluss auf den *Verlust* (eng.: *loss*) als bei der \mathcal{L}_1 Funktion. Diese Verlustfunktion wird auch als *mittlere quadratische Abweichung* (engl.: *mean squared error*) bezeichnet.

$$\mathcal{L}_2 = (\hat{y} - y)^2 \quad (2.4)$$

Binary Crossentropy Loss Die Binary Crossentropy Verlustfunktion ist auch als *logarithmische Verlustfunktion* bekannt. Sie eignet sich für binäre Klassifikationen, wo demnach das KNN eine Wahrscheinlichkeit zwischen 0 und 1 ausgibt. Ein Beispiel hierfür ist, wenn das KNN abschätzen soll, ob auf einem Bild eine Katze zu sehen ist oder nicht. Ein Schwellenwert für eine Wahrscheinlichkeit zwischen 0 und 1 gibt dann an, ob das Ergebnis als *Ja* oder *Nein* interpretiert wird. Die Binary Crossentropy Verlustfunktion ist in Gleichung 2.5 dargestellt. Wenn der erwartete Wert y gleich 1 ist, dann reduziert sich die Verlustfunktion zu $\mathcal{L} = -\log(\hat{y})$. Ist der erwartete Wert y gleich 0, dann reduziert sich die Verlustfunktion zu $\mathcal{L} = -\log(1 - \hat{y})$. Die Verlustfunktion nutzt die Eigenschaften aus, dass $\log(1) = 0$ ist und der Logarithmus von Werten zwischen 0 und 1 negativ ist. [10]

$$\mathcal{L}_{BCE} = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})] \quad (2.5)$$

Das Training des Modells besteht nicht nur daraus, die Vorhersagen des Modells zu bewerten. Damit das Modell in dem nächsten Trainingsdurchlauf idealerweise einen geringeren Wert für die Kostenfunktion erreicht, müssen die trainierbaren Parameter θ des Modells angepasst werden. Von ihnen hängt der Wert der Kostenfunktion ab. Die Ausgangssituation ist hierbei folgende: Mit einem gegebenen θ befindet sich das KNN in einem bestimmten Punkt der Kostenfunktion $J(\theta)$. Gesucht ist eine Parameteränderung $d\theta$, mit der sich das KNN am weitesten an das globale Minimum der Kostenfunktion annähert. Das globale Minimum ist die beste Lösung, die das Modell für die gegebenen Trainingsdaten finden kann und damit das Optimum. Da $J(\theta)$ eine viel-dimensionale Funktion ist, berechnet die Trainingsfunktion diesen Idealwert für θ nicht numerisch. Stattdessen nähert sich das KNN mit jedem Trainingsschritt dem Optimum an. [8]

Nähern tut sich das KNN dem Optimum, indem es den Punkt θ in Richtung des negativen Gradienten der Kostenfunktion bewegt. Also die Richtung, in die, aus der momentanen Ausgangsposition, die Kostenfunktion den steilsten Abstieg besitzt. Pro Trainingsiteration bewegen sich die Parameter θ nur um einen kleinen Betrag in Richtung des negativen Gradienten. Das wird so lange durchgeführt, bis der Gradient einen so geringen Betrag hat, dass sich die Parameter des KNN nicht mehr signifikant verändern. Das KNN befindet sich hier bestenfalls im globalen Optimum. Der Betrag der Annäherung pro Trainingsschritt ist durch eine sogenannte

Lernrate α bestimmt. Die Lernrate ist ein *Hyperparameter*, und damit klassischerweise ein nicht-trainierbarer Parameter, da sie durch die Entwickler fest bestimmt wird und nicht durch das Modell selbst gelernt wird. [8]

Auf Ableitung der Kostenfunktion für Berechnung des Gradienten eingehen?

Hier schon Batches erklären und damit darauf eingehen wie mit einer Verlustfunktion über mehrere Trainingsbeispiele trainiert wird

2.2.2 Convolutional Neural Networks

KNNs bewähren sich mitunter besonders im Bereich *Computer Vision*. Ein Bereich, der sich mit der Interpretation von Bild- und Videodaten beschäftigt. Hier spielt die Mustererkennung eine tragende Rolle. Es sollen Merkmale erkannt werden, die jedes Objekt eines bestimmten Typs auszeichnen, die jedoch nicht auf jedem Bild die exakt identischen Pixelwerte besitzen. Die typische Form von Katzenohren ist beispielsweise ein Muster, das bei der Katzenerkennung verwendet werden kann. Es ist nicht trivial, allgemeine Regeln zu definieren, welche Pixelmuster als Katzenohr erkannt werden sollen und welche nicht. Deshalb wird hier auf KNNs zurückgegriffen.

Verwendet man hierfür jedoch die bisher beschriebene Netzwerkarchitektur, treten verschiedene Probleme auf. Jedes sogenannte *Feature* des Eingangs wird über die Eingangsschicht in das KNN gespeist. Bei einem Schachalgorithmus kann die Menge aller Features beispielsweise durch die momentane Position aller Figuren auf dem Schachbrett beschrieben werden. Das liegt daran, dass genau diese Werte den Ausgang des Netzwerks beeinflussen. In diesem Fall, welchen Zug der Algorithmus als nächstes spielt. Bei der Bildklassifizierung ist jeder Pixel des Bildes ein Feature. Ein Netz, das Bilder der Größe 1024x1024 Pixel mit drei Farbkanälen (rot, grün blau) klassifizieren soll, muss demnach folgende Anzahl an Eingängen verarbeiten:

$$1024 \cdot 1024 \cdot 3 = 3.145.728 \quad (2.6)$$

Um eine derartige Anzahl an Eingängen sinnvoll interpretieren zu können, ist ein Netzwerk mit vielen Schichten und Neuronen notwendig. So vielen, dass auch heutige Computer an die Grenzen ihrer Rechenleistung gelangen. Mitunter deshalb wird im Bereich Computer Vision auf Convolutional Neural Networks (CNNs) zurückgegriffen.

CNNs basieren auf der Faltung (engl.: convolution) einer Eingangsmatrix mit einer Faltmatrix. Jedes CNN besitzt mindestens eine Schicht, die eine Faltung durchführt. Dabei schiebt das CNN die Faltmatrix nach und nach über die Eingangsmatrix. Bei jedem Schritt berechnet es dabei

das Skalarprodukt der momentan betrachteten Werte der Eingangsmatrix mit den Parametern der Faltmatrix. Das Ergebnis hiervon ist eine neue Matrix, die weniger Werte beinhaltet als der Eingang des CNN. Folgt hierauf eine weitere Schicht, dann erhält sie das Ergebnis der Faltung als Eingabewert und nutzt eine eigene Faltmatrix um erneut eine Faltung durchzuführen. Die trainierbaren Parameter sind dabei die Werte der Faltmatrizen aller Schichten. Nachfolgende Abbildung soll die Faltung visualisieren:

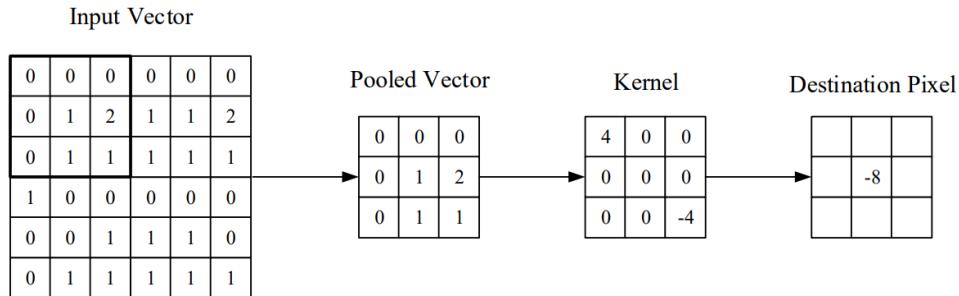


Abbildung 2.5: Beispiel für eine Faltung (engl.: Convolution) [11]

Zunächst wird die Faltmatrix beispielsweise *links oben* auf der Eingangsmatrix angewendet. Der betrachtete Teil der Eingangsmatrix wird hier als *Pooled Vector* bezeichnet. Hierauf wird der Kernel, in diesem Fall ein sogenannter *3x3-Kernel*, angewendet. Es wird also das Skalarprodukt der Werte mit dem gleichen Index aus dem *Pooled Vector* und dem Kernel berechnet. In diesem Fall:

2.3 Bildgenerierung mittels Künstlicher Neuronaler Netze

Der Lernfortschritt von Klassifikatoren besteht darin, besser in der Aussage zu werden, ob eine Vorhersage y auf einen gegebenen Eingang x zutrifft. Das bezeichnet man auch als *diskriminative Modellierung*. Um das zu erlernen, benötigt der Klassifikator annotierte Trainingsdaten. Das bedeutet, dass jedes Trainingsbeispiel x eine erwartete Vorhersage y besitzt. Man bezeichnet das auch als *überwachtes Lernen*. [12]

Neben der diskriminativen Modellierung ist auch eine *generative Modellierung* möglich. Generative KNNs sollen lernen neue Daten zu erzeugen, die der Verteilung der Trainingsdaten ähneln. Dazu erlernen sie die statistische Verteilung der Trainingsdaten. Generative Netze zur Bilderzeugung können demnach Bilder generieren, die den Trainingsbildern ähnlich sehen. Die Trainingsbilder sind nicht annotiert, wodurch generative Netze in der Regel in das *unüberwachte Lernen* einzuordnen sind. [12]

2.3.1 Mathematischer Hintergrund

Generative Netze zur Bilderzeugung sollen beurteilen können, wie wahrscheinlich es ist, dass ein gegebenes Bild aus der Verteilung der Trainingsdaten stammt. Wenn x für jedes mögliche existierende Bild steht, so bilden generative Netze folgende Wahrscheinlichkeitsverteilung ab: [12]

$$\hat{p}(x) \quad (2.7)$$

Für ein gegebenes Bild x gibt $\hat{p}(x)$ einen Schätzwert dafür an, wie wahrscheinlich es ist, dass das Bild aus den Trainingsdaten stammt. Diese Wahrscheinlichkeitsverteilung wird durch das Netz erlernt. Optimiert wird, dass die geschätzte Verteilung der Daten $\hat{p}(x)$ möglichst ähnlich zu der tatsächlichen Verteilung der Trainingsdaten $p(x)$ ist. Ein Beispielhafter Vergleich ist in Abbildung 2.6 dargestellt. Es ist erkennbar, dass sich die geschätzte und die tatsächliche Verteilung ähnlich sehen, jedoch nicht identisch sind. Die Abweichung zwischen diesen Verteilungen stellt dabei die Kosten (engl.: den *loss*) dar. Die schwarzen Punkte kennzeichnen Trainingsdaten. Durch sie soll die Verteilung $p(x)$ abgebildet werden. Weniger diversifizierte Trainingsdaten würden sich beispielsweise nur in einem Teilbereich von $p(x)$ befinden. Dadurch könnte das Modell $p(x)$ weniger gut approximieren. [12] [13]

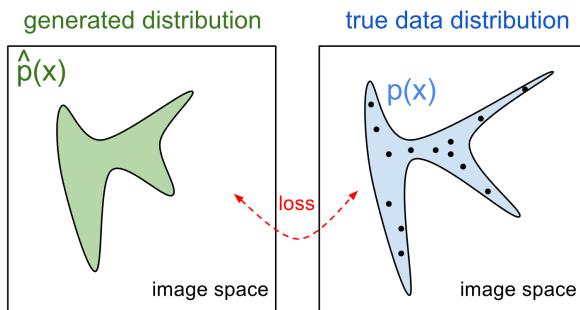


Abbildung 2.6: Beispielhafter Vergleich von $\hat{p}(x)$ und $p(x)$ [13]

Bei der Bildgenerierung versucht das Netz den Wahrscheinlichkeitswert für $\hat{p}(x)$ zu maximieren. Es lernt durch $\hat{p}(x)$, wie die Verteilung der Trainingsdaten aussieht und versucht anschließend ausschließlich Bilder zu generieren, die dieser Verteilung folgen. Bezogen auf Abbildung 2.6 befinden sich alle generierten Bilder des trainierten Netzes im grün markierten Bereich. [12] [13]

Es existieren verschiedene Arten generativer Netze. Die Taxonomie, also die Einteilung verschiedener Netze in bestimmte Kategorien, kann Abbildung 2.7 entnommen werden. Einerseits existieren Architekturen, die die Wahrscheinlichkeitsverteilung $\hat{p}(x)$ explizit berechnen. Andere berechnen die Funktion nicht, verwenden sie jedoch implizit. In der Abbildung wird dahingehend zwischen *Explicit Density Models* (*explizit berechnete Dichte*) und *Implicit Density*

Models (implizit berechnete Dichte) unterschieden. Es existieren zudem Unterkategorien, mittels derer eine feinere Kategorisierung durchgeführt wird. [14]

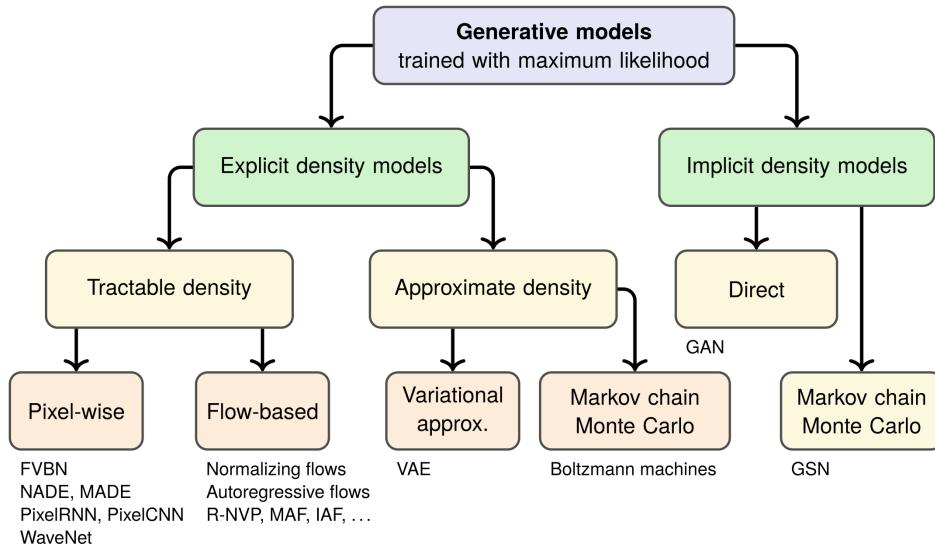


Abbildung 2.7: Taxonomie generativer Modelle [14]

Grafik ersetzen durch deutsche Taxonomie?

Es soll in diesem Kapitel auf verschiedene Architekturen generativer Netze eingegangen werden.

2.3.2 Pixel Recurrent Neural Networks

Die Architektur der Pixel Recurrent Neural Networks (PixelRNNs) stammt aus dem Jahr 2016. Diese Netze stützen sich explizit auf die Maximierung der Maximum-Likelihood-Schätzung von $\hat{p}(x)$ für jeden Pixel. Sie sind in der genannten Taxonomie den Modellen zuzuordnen, die den tatsächlichen Schätzwert von $p(x)$ berechnen können. [15]

Im folgenden soll geklärt werden, wie der optimale Wert für jeden Pixel eines generierten Bildes bestimmt wird. Ein betrachtetes Bild x der Auflösung $n \times n$ kann in seine einzelnen Pixel $(x_1, x_2, \dots, x_{n^2})$ aufgeteilt werden. In Gleichung 2.8 ist dabei dargestellt, wie die Wahrscheinlichkeit eines jeden Pixels in die gesamte Verteilung $\hat{p}(x)$ einfließt. [15]

$$\hat{p}(x) = \hat{p}(x_1, x_2, \dots, x_{n^2}) = \prod_{i=1}^{n^2} \hat{p}(x_i | x_1, \dots, x_{i-1}) \quad (2.8)$$

Jeder Pixel x_i besitzt eine eigene Wahrscheinlichkeitsverteilung $\hat{p}(x_i|x_1, \dots, x_{i-1})$. Sie ist abhangig von allen anderen Pixeln x_1, \dots, x_{i-1} des Bildes. Der absolut optimale Wert eines Pixels kann

demnach nur dann berechnet werden, wenn die Werte aller anderen Pixel bekannt sind. Das Produkt aller Wahrscheinlichkeitswerte der einzelnen Pixel ergibt $\hat{p}(x)$. Soll $\hat{p}(x)$ maximiert werden, so müssen die Terme $\hat{p}(x_i|x_1, \dots, x_{i-1})$ möglichst hohe Werte liefern. Daraus ergibt sich dann unter gegebenem Kontext für jeden Pixel eine Maximum-Likelihood-Schätzung. Also der Wert, für den $\hat{p}(x)$ möglichst weit gegen *eins* strebt. [15]

Die Idee von PixelRNNs ist, dass bei der Generierung in einer Ecke des Bildes gestartet wird. Das Bild wird zunächst auf einen Pixel reduziert, der im folgenden x_1 genannt wird. Für diesen Pixel wird ein Wert generiert. Anschließend wird x_1 gemeinsam mit einem benachbarten Pixel x_2 betrachtet. Die Wahrscheinlichkeitsverteilung für x_2 ergibt sich dadurch zu $\hat{p}(x_2|x_1)$. Der Wert für x_2 ist somit nur von x_1 abhängig. Da x_1 bekannt ist, kann ein optimaler Wert für x_2 bestimmt werden. Die Wahrscheinlichkeitsverteilung von x_3 ergibt sich zu $\hat{p}(x_3|x_1, x_2)$, die von x_4 zu $\hat{p}(x_4|x_1, x_2, x_3)$. Das Bild wird sukzessive generiert, wobei der momentane Pixelwert für x_i von allen bisher generierten Pixeln abhängig ist. Dieses vorgehen ist in Abbildung 2.8 dargestellt. Der Wert des rot markierten Pixels hängt von allen blau markierten Pixel ab. Ist für diesen ein Wert bestimmt, wird der rechtsseitig benachbarte Pixel als neues x_i gewählt.

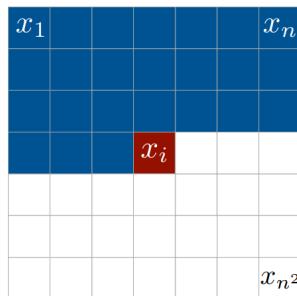


Abbildung 2.8: Bestimmung von $\hat{p}(x)$ mit PixelRNNs [15]

Um die beschriebene Abhängigkeit des momentan generierten Pixels zu allen bisher generierten Pixel umsetzen zu können, besitzen PixelRNNs eine Art *Erinnerung*. Bisher wurden in dieser Arbeit nur sogenannte *Feedforward* Netze behandelt, bei denen der Informationsfluss stets in eine Richtung erfolgt. Nämlich vom Eingang des Netzes zum Ausgang. Es existieren auch *Recurrent Neural Networks*. Sie werden besonders zur Verarbeitung von natürlicher Sprache eingesetzt (engl.: natural language processing). Abbildung 2.9 bildet hierfür eine beispielhafte Darstellung.

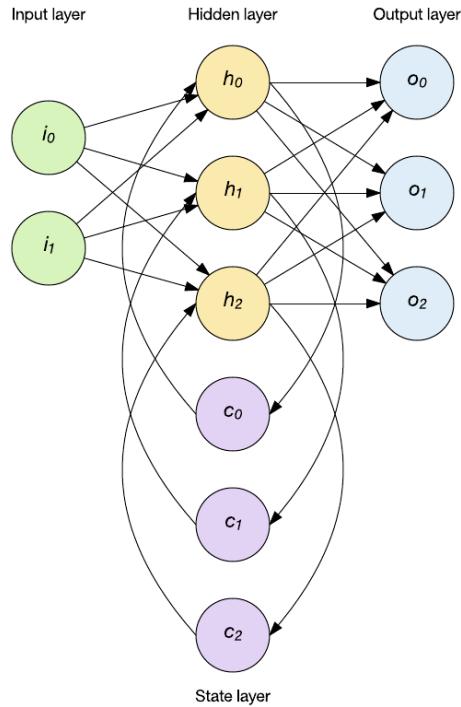


Abbildung 2.9: Darstellung eines Recurrent Neural Networks [16]

In Recurrent Neural Networks spielen die *Zustände* eines Netzes eine besondere Rolle. Ein Zustand wird durch die Eingangs- und Ausgangswerte aller Neuronen zu einem gegebenen Zeitpunkt beschrieben. In PixelRNNs ist der Zustand des Netzes für den Pixel x_2 abhängig von dem Zustand des Netzes für x_1 . Um solche Beziehungen darstellen zu können, besitzt das beispielhaft abgebildete Netz die Neuronen c_0 bis c_2 , die den vorherigen Wert eines Neurons rekursiv auf seinen Eingang zurückführen. Somit wird der vorherige Zustand des neuronalen Netzes als zusätzlicher Eingang für die Berechnungen genutzt. PixelRNNs nutzen eine besondere Form der Recurrent Neural Networks. Sie arbeiten mit sogenannter *Long Short-term Memory*. Dadurch soll das Problem behoben werden, dass in klassischen Recurrent Neural Networks weit in der Vergangenheit liegende Zustände nur noch einen geringen Einfluss auf den momentanen Zustand haben. [17]

Da die durch ein PixelRNN umgesetzte Verteilung $\hat{p}(x)$ direkt erfassbar ist, wird ihnen nachgesagt, dass die Performanz solcher Netze gut evaluiert werden kann. Es gilt als vergleichsweise leicht, für solche Netze Metriken zur Messung der Performanz umzusetzen. Ein grundlegender Nachteil von PixelRNNs ist, dass die Generierung sequenziell erfolgt. Es ist in dem beschriebenen Verfahren nicht möglich, mehrere Pixel parallel zu generieren, da der Wert eines Pixels von denen aller vorher generierten Pixel abhängig ist. Dies verlangsamt die Generierung, da keine Parallelisierung möglich ist. [17]

Es existieren auch sogenannte *PixelCNNs*, bei denen sich die Berechnung stets nur auf bestimmte Bildbereiche konzentriert. Diese Bildbereiche können parallel zueinander sequenziell berechnet

werden. Die Parallelisierung ist jedoch nur während des Trainings des Netzwerks oder während der Evaluation von $\hat{p}(x)$ für gegebene Bilder möglich. Die Bildgenerierung erfolgt auch hier, analog zu PixelRNNs, vollständig sequenziell. [15]

2.3.3 Autoencoder

Das Ziel von Autoencodern ist, den Eingang des Netzes am Ausgang zu rekonstruieren. Dazu setzen sich diese Netze aus drei Bestandteilen zusammen: dem Kodierer, dem latenten Raum und dem Dekodierer. In Abbildung 2.10 ist eine beispielhafte Autoencoderarchitektur dargestellt.

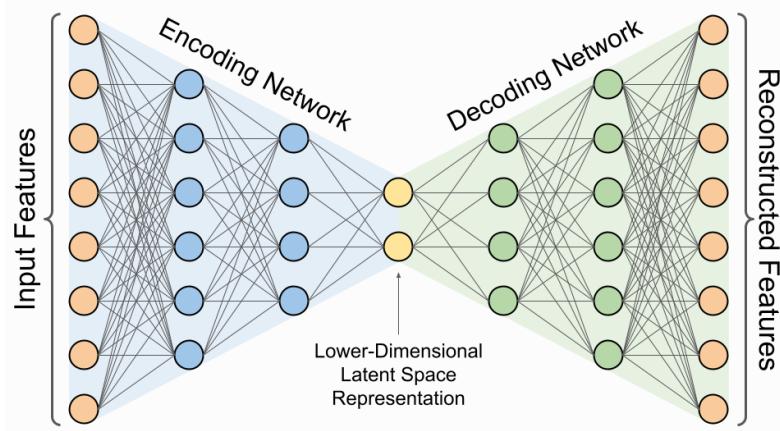


Abbildung 2.10: Architektur eines Autoencoders

Der **Kodierer** besitzt die Aufgabe, Merkmale aus dem Eingang des Netzes zu extrahieren. Diese Merkmale sollen daraufhin mit einer begrenzten Anzahl an Parametern durch den latenten Raum repäsentiert werden. Somit besteht die Aufgabe des Kodierers darin, den Eingang auf seine für das Netz wesentlichen Eigenschaften zu reduzieren. Und zwar erfolgt die Komprimierung dabei so, dass die Informationen gerade so durch den latenten Raum dargestellt werden können.

Bei dem **latenten Raum** handelt es sich um eine einzelne Schicht von Parametern, oder in diesem Kontext: Neuronen. Je mehr Neuronen sich in dem latenten Raum befinden, desto mehr Informationen können von der Kodierung an die Dekodierung übertragen werden. Die Merkmale, die sich in dem latenten Raum befinden, können durch einen Vektor \vec{z} beschrieben werden. Der latente Raum sollte klein genug sein, damit dort nicht alle Merkmale des Eingangs gespeichert werden können. So wird der Kodierer dazu gezwungen, vereinzelte Merkmale zu extrahieren.

Der **Dekodierer** nutzt die Werte aus dem latenten Raum, um den Eingang nachzubilden. Diese Nachbildung stellt den Ausgang des Autoencoders dar. Die Kosten eines Autoencoders können durch die Abweichung zwischen Ein- und Ausgang festgestellt werden.

Da der Zweck von Autoencodern darin besteht, einen Eingang auf seine relevanten Merkmale zu reduzieren, wird der Dekodierer nur während des Trainings verwendet. Beim praktischen Einsatz eines Autoencoders werden lediglich der Kodierer und der latente Raum eingesetzt. Im Gegensatz zu PixelRNNs basieren Autoencoder klassischerweise nicht auf Recurrent Neural Networks, sondern auf Feedforward Netzen.

Diese beschriebene Architektur der Autoencoder ist nicht für die generative Modellierung geeignet, da sie deterministisch ist. Erhält das Modell bestimmte Eingangswerte, so liefert es stets die gleichen Ausgangswerte. Es versucht den Eingang möglichst zu rekonstruieren, wobei der Inhalt \vec{z} des latenten Raums für ein gegebenes Eingangsbild stets gleich ist. Eine zufällige Erzeugung neuer Bilder ist damit nicht möglich. Die sogenannten *Variational Autoencoder* sind eine Architektur, die für die Generierung neuer Daten verwendet werden können. [9]

Variational Autoencoder verfolgen das Ziel, eine Zufallskomponente in die Bildzeugung einfließen zu lassen. Ein entscheidender Unterschied zu klassischen Autoencodern ist deshalb der folgende: Ein gegebener Eingang x wird auf kein festes \vec{z} kodiert, sondern auf eine Wahrscheinlichkeitsverteilung. Sie wird bezeichnet als:

$$p(z|x) \quad (2.9)$$

Im Gegensatz zu PixelRNNs versucht das Netz somit nicht die Verteilung der Trainingsdaten $p(x)$ zu approximieren, sondern die Verteilung der Merkmale \vec{z} der Trainingsdaten x . Es wird angenommen, dass jedes Merkmal normalverteilt ist. Damit kann jede Komponente z_i des Vektors $\vec{z} = [z_1, z_2, \dots, z_n]^T$ durch eine Gaußsche Normalverteilung $\mathcal{N}(\mu_i, \sigma_i^2)$ beschrieben werden. Aufgabe des Kodierers ist damit nicht mehr, aus einem gegebenen x eine Menge von Merkmalen \vec{z} zu bestimmen. Stattdessen soll der Kodierer die Vektoren $\vec{\mu}$ und $\vec{\sigma}$ bestimmen, durch die sich die einzelnen Normalerteilungen von \vec{z} beschreiben lassen.

An den Dekodierer wird ein zufällig aus der Verteilung $p(z|x)$ entnommenes Set an Merkmalen \vec{z} übergeben. Der Dekodierer übersetzt dieses gegebene \vec{z} daraufhin in ein Bild. Im praktischen Einsatz werden bei einem Variational Autoencoder nur der latente Raum und der Dekodierer genutzt. Der Dekodierer erhält zufällige Werte für \vec{z} , also zufällige Merkmale, und generiert daraus ein Bild.

Vor- Nachteile?

[18]

2.3.4 Generative Adversarial Networks

Eine weitere Architektur, die zur Bildgenerierung verwendet werden kann, sind sogenannte *Generative Adversarial Networks* (GANs). Sie wird in einer Arbeit aus dem Jahre 2014 erstmals vorgestellt [19]. Zudem existiert eine Veröffentlichung aus dem Jahre 2020 [20]. Während in der ersten Veröffentlichung Generative Adversarial Networks (GANs) als eine neuartige Architektur vorgestellt werden, zeigt die neuere Arbeit vor allem Erfolge und Hindernisse auf, die sich im Laufe der Zeit bei der Verwendung von GANs herausgestellt haben.

Ein GAN besteht aus zwei Komponenten: Dem *Generator* und dem *Diskriminator* (engl.: *Discriminator*). Der Generator erzeugt aus einem zufälligen Eingangsvektor ein Bild. Der Diskriminator erhält ein Bild als Eingang und soll bewerten, ob das Bild echt oder künstlich generiert ist. Bei beiden Komponenten handelt es sich um KNNs. Das Ziel des Trainings ist, dass der Generator Bilder erzeugen kann, die der Diskriminator nicht von echten Trainingsbildern unterscheiden kann. Dabei wird der Generator besser in seiner Generierung, während der Diskriminator besser in seiner Unterscheidung wird. Mit zunehmender Güte des Generators muss auch der Diskriminator weitere Merkmale erlernen, anhand derer er künstliche Bilder erkennt. Dies gilt ebenso in die entgegengesetzte Richtung. Damit agieren Generator und Diskriminator als direkte Gegenspieler, die versuchen einander zu überlisten. [20]

Die Güte des Generators ist dadurch gegeben, wie ähnlich die von ihm erzeugte Wahrscheinlichkeitsverteilung \hat{p} zu der Verteilung der Trainingsdaten p ist. Der Diskriminator wird hingegen darin bewertet, wie erfolgreich er in seiner Aussage ist, ob ein gegebenes Bild entweder \hat{p} oder p stammt. [20]

Das Zusammenspiel zwischen Generator und Diskriminator während des Trainings ist in Abbildung 2.11 dargestellt.

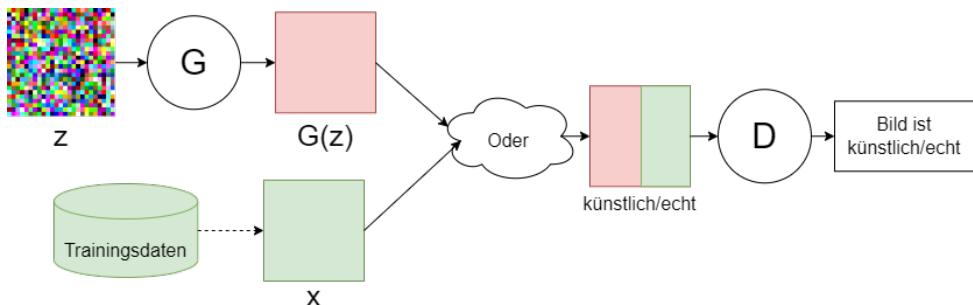


Abbildung 2.11: Zusammenspiel zwischen Generator und Diskriminator

Der Eingangsvektor heißt z . Daraus erzeugt Generator G ein Bild $G(z)$. Der Diskriminator D erhält entweder ein Bild x aus den Trainingsdaten oder ein Bild $G(z)$ vom Generator. Darauf basierend trifft der Diskriminator entweder eine Vorhersage $D(x)$ oder $D(G(z))$, wie wahrscheinlich es ist, dass das Bild real ist. Bei der Vorhersage handelt es sich um eine Wahrscheinlichkeit

zwischen 0 und 1. Ein Wert von 0.8 bedeutet demnach zum Beispiel: Der Diskriminator ist sich zu 80% sicher, dass das Bild real ist. [20]

Training Hieraus ergibt sich die Kostenfunktion für GANs. Sie nennt sich $V(D, G)$: [19]

$$\min_G \max_D V(D, G) = \mathbb{E}_x[\log(D(x))] + \mathbb{E}_z[\log(1 - D(G(z)))] \quad (2.10)$$

Bei dem Training handelt es sich um ein Min-Max-Problem. Der Generator versucht die Funktion $V(G, D)$ zu minimieren, wohingegen der Diskriminator sie zu maximieren versucht. Das Ziel des Trainings ist damit nicht wie klassischerweise bei KNNs, dass die Funktion einen Wert von 0 anstrebt. Stattdessen soll sie im Verlauf des Trainings gegen einen Wert größer als 0 konvergieren. Hier können sich der Generator und der Diskriminator nicht weiter verbessern. Sie haben idealerweise optimale Parameter. Es wird davon gesprochen, dass G und D sich in einem *Nash-Gleichgewicht* befinden. Dieser Begriff stammt aus der Spieltheorie und kann deshalb verwendet werden, weil ein GAN als ein Spiel zwischen Generator und Diskriminator beschrieben werden kann. [20]

Die Funktion $V(D, G)$ basiert auf einer logarithmischen Verlustfunktion. Der Term 2.11 aus der Kostenfunktion stellt die Fehlerrate des Diskriminators auf den echten Trainingsdaten dar. Die Fragestellung lautet: *Wenn ich ein zufälliges Bild x aus den Trainingsdaten ziehe, wie wahrscheinlich ist es, dass der Diskriminator es als echt klassifiziert?* Der Wert für $\log(D(x))$ sollte möglichst nahe 0 sein, da für jedes echte Trainingsbild x zu erwarten ist, dass $D(x)$ einen Wert nahe 1 ausgibt.

$$\mathbb{E}_x[\log(D(x))] \quad (2.11)$$

Der Term 2.12 beschreibt hingegen, wie viele generierte Bilder der Diskriminator als echt klassifiziert. Hierfür werden zufällige z aus der Wahrscheinlichkeitsverteilung der Eingangsvektoren entnommen. Diese Verteilung kann beliebig definiert sein. Ist der Diskriminator besser trainiert als der Generator, dann ist zu erwarten, dass $D((G(z)))$ einen Wert nahe 0 hat. Somit ist der Logarithmus ebenfalls nahe 0. Im umgekehrten Fall nimmt der Logarithmus einen negativen Wert an. Der Betrag des Werts erhöht sich, je mehr $D(G(z))$ gegen 1 strebt.

$$\mathbb{E}_z[\log(1 - D(G(z)))] \quad (2.12)$$

Der Term 2.11 ist nicht vom Generator abhängig, sondern lediglich von D und x . Hierauf hat der Generator keinen Einfluss, wodurch er alleine durch diesen Teil der Kostenfunktion nicht trainiert wird. Der Diskriminator besitzt somit zwei Terme, die ihn trainieren, während der Generator nur einen besitzt. Aus diesem Grund werden dem Diskriminator in der Regel doppelt

so viele unechte Daten wie echte Trainingsdaten gezeigt. Dies soll einem ungleichen Training der beiden Komponenten des GANs entgegenwirken. Der Optimalzustand eines GANs ist, dass der Diskriminator so gut wie möglich identifizieren kann, ob ein gegebenes Bild aus $p(x)$ stammt, während der Generator dennoch in der Lage ist, den Diskriminator zu überlisten. Das Training von GANs gilt als empfindlich gegenüber den gewählten Hyperparametern und der Netzwerkarchitektur. GANs wird nachgesagt, dass kleine Änderungen in den beiden genannten Aspekten die Qualität der Generierten Bildern signifikant beeinflussen können. [9]

Im praktischen Einsatz wird nur der Generator des GANs verwendet. Der Diskriminator wird ausschließlich dazu eingesetzt, mit $p(G(z))$ möglichst gut $p(x)$ zu approximieren, sodass die generierten Bilder im Optimalfall nicht von echten Trainingsdaten zu unterscheiden sind. [9]

Die bisher beschriebene Architektur von GANs wird auch als *Vanilla GAN* bezeichnet. Forscher und Anwender haben seit dieser Veröffentlichung verschiedene Limitationen und Probleme bei Vanilla GANs feststellen können. Insbesondere im Hinblick auf spezielle Einsatzgebiete. Ein hier häufig anzutreffender Begriff ist *Modal Collaps*. Damit ist die Situation gemeint, dass der Generator bei beliebigem Input stets dasselbe Bild generiert. Er lernt, dass ein bestimmtes Bild den Diskriminator überlisten kann und generiert es deshalb jedes Mal, egal welchen Input man ihm zuführt. In dem Anwendungsfall dieser Arbeit könnte das zum Beispiel dazu führen, dass der Generator nur eine einzige Art von Straßenschild generiert. Lösen lässt sich das Problem des *Modal Collaps* beispielsweise mit sogenannten CycleGANs. [9]

CycleGANs CycleGANs eignen sich für eine bestimmte Art der generativen Modellierung: Der Bild-zu-Bild Übersetzung. Hierbei soll das Modell nicht ein völlig neues Bild generieren, sondern ein vorhandenes Bild in eine andere Domäne übersetzen. Ein Beispiel dafür ist das umwandeln von Zeichnungen in fotorealistische Bilder oder das einfärben von schwarz-weiß Bildern. Ein entscheidender Unterschied ist somit, dass der Generator keinen zufälligen Vektor z als Eingang enthält, sondern ein Bild, das in eine andere Domäne übersetzt werden soll. Bei solchen Anwendungsfällen sollen CycleGANs einen Modal Collaps verhindern. Das Ziel ist, dass das erzeugte Bild des Modells immer abhängig von dem Eingangsbild ist. [21]

Ein CycleGAN besteht aus zwei miteinander gekoppelten GANs. Generator G übersetzt ein Bild X in ein Bild Y . Aus diesem Y soll dann Generator F den Eingang X rekonstruieren. Somit soll das gesamte Netzwerk nicht nur neue Bilder generieren können, sondern soll auch von einem generierten Bild zurück auf den zugeführten Eingang schließen können. Dies lässt sich mathematisch so darstellen, dass G und F folgende Abbildungen implementieren: [21]

$$\mathbf{G} : X \mapsto Y \wedge \mathbf{F} : Y \mapsto \tilde{X} \quad (2.13)$$

Das Modell G erzeugt somit aus einem gegebenen X ein Y , wohingegen F aus dem Y auf das X schließen soll. Die Behebung des Modal Collaps findet dadurch statt, dass das Netzwerk den Output \tilde{X} von F überprüft und diesen mit dem tatsächlichen Eingang X vergleicht. Es wird überprüft, wie ähnlich sich \tilde{X} und X sind. Liegt eine zu hohe Diskrepanz vor, kann das Netzwerk darauf schließen, dass G Outputs erzeugt, die nicht in direkter Abhängigkeit zu X stehen. [21]

Das Training von CycleGANs basiert auf mehreren Kostenfunktionen. Die GANs G und F besitzen jeweils eigene Diskriminatoren D_Y und D_X . Dadurch haben beide GANs einen eigenen *Adversarial Loss*. Damit ist die in Gleichung 2.10 beschriebene Kostenfunktion eines Vanilla GANs gemeint. Im Kontext von CycleGANs ist sie, für das GAN G wie folgt definiert: [21]

$$\mathcal{L}_{GAN}(G, D_Y, X, Y) = \mathbb{E}_y[\log D_Y(y)] + \mathbb{E}_x[\log(1 - D_Y(G(x)))] \quad (2.14)$$

Die Funktion für GAN F ist identisch, mit dem Unterschied, dass sie von F und D_X statt von G und D_Y abhängt. Ein Unterschied zu den Vanilla GANs ist hierbei: Bei Vanilla GANs steht der Buchstabe x für die Trainingsdaten aus der Zieldomäne. Hier sind jedoch einzelne x die Eingabewerte für das CycleGAN. Was vorher z war, ist demnach hier x . Die Trainingsdaten aus der Zieldomäne werden mit dem Formelzeichen y abgekürzt. [21]

Als weitere Kostenfunktion besitzen CycleGANs einen *Cycle Consistency Loss* (Gleichung 2.15). Die beiden Summanden der Gleichung setzen sich daraus zusammen, wie weit die Pixelwerte von den generierten Bildern und den echten Bildern auseinander liegen. Mit $G(x)$ wird ein Bild \tilde{y} generiert, F generiert anschließend aus diesem \tilde{y} wieder ein \tilde{x} . Wenn \tilde{x} und x möglichst ähnlich sind, dann kann davon ausgegangen werden, dass die generierten Bilder des Netzwerks G in direkter Abhängigkeit von dem Input X stehen. Was hierbei berechnet wird, ist die durchschnittliche, absolute Abweichung der Pixelwerte. [21]

$$\mathcal{L}_{cyc}(G, F) = \mathbb{E}_x[\mathcal{L}_1(F(G(x)) - x)] + \mathbb{E}_y[\mathcal{L}_1(G(F(y)) - y)] \quad (2.15)$$

Um die gesamten Kosten des CycleGANs zu erhalten, werden die bisher beschriebenen Kostenfunktionen addiert. Der Cycle Consistency Loss $L_{cyc}(G, F)$ wird dabei mit einem absoluten Wert λ multipliziert, um die Kosten dieser Funktion im Vergleich zu den *Adversarial Losses* gewichten zu können. In der Veröffentlichung der CycleGANs wird ein λ von 10 verwendet. Somit wird mit einem vergleichsweise hohen Gewicht versehen, dass die generierten Bilder in direkter Abhängigkeit zu den Eingangswerten des CycleGANs stehen. Der Wert λ stellt einen Hyperparameter dar. [21]

$$\mathcal{L}(G, F, D_X, D_Y) = \mathcal{L}_{GAN}(G, D_Y, X, Y) + \mathcal{L}_{GAN}(F, D_X, Y, X) + \lambda \cdot \mathcal{L}_{cyc}(G, F) \quad (2.16)$$

[21]

Einige Implementierungen teilen die Kostenfunktion auf einzelne Verluste für jeweils die Generatoren G und F sowie die Diskriminatoren D_y und D_x auf: [22] [23]

$$\mathcal{L}_G = \mathbb{E}_x[\log(D_Y(G(x)))] + \mathcal{L}_{cyc}(G, F) \quad (2.17a)$$

$$\mathcal{L}_F = \mathbb{E}_y[\log(D_X(F(y)))] + \mathcal{L}_{cyc}(F, G) \quad (2.17b)$$

$$\mathcal{L}_{D_y} = 0.5 \cdot \mathcal{L}_{GAN}(G, D_Y, X, Y) \quad (2.17c)$$

$$\mathcal{L}_{D_x} = 0.5 \cdot \mathcal{L}_{GAN}(F, D_X, Y, X) \quad (2.17d)$$

Das erlaubt ein separates Training der einzelnen KNNs des CycleGANs. Die Generatoren besitzen in ihrer Kostenfunktion lediglich den Teil des Adversarial Loss, den sie beeinflussen können. Dieser ist jedoch so abgeändert, dass die Generatoren einen möglichst hohen Wert für $D_y(G(x))$ beziehungsweise $D_x(F(y))$ anstreben. In diesem Fall führt dort ein Wert nahe 1 zu Kosten von Nähe 0. Das entspricht dem, dass die Generatoren versuchen, den entsprechenden Diskriminator zu Falschaussagen zu verleiten.

Die Qualität der Diskriminatoren wird anhand des gesamten Adversarial Loss gemessen. Diese Kosten werden bei den Diskriminatoren mit 0.5 multipliziert, damit die Diskriminatoren nicht schneller trainiert werden als die Generatoren [22]. Stattdessen wäre es auch möglich, die Trainingschritte der Generatoren doppelt so oft durchzuführen, wie die der Diskriminatoren [9].

Die Basis für G und F ist eine bestimmte CNN-Architektur, die sich Residual Neural Network (ResNet) nennt. Diese wurde im Jahre 2015 von Forschern bei Microsoft Research eingeführt. Die Autoren der CycleGAN Veröffentlichung wählen diese Architektur, weil sie in einem ähnlichen Anwendungsfall bei anderen Autoren überzeugende Resultate gezeigt habe. [24] [21]

Erklären, was ein Residual Block ist.

Conditional GANs Eine weitere Möglichkeit, die Generierung von GANs zu steuern, sind sogenannte Conditional Generative Adversarial Networks (cGANs). Dabei erhält der Generator zusätzlich zu dem zufällig erzeugten Eingangsvektor z noch eine Bedingung x . Dabei kann es sich zum Beispiel um eine Zahl handeln, die für ein bestimmtes Objekt steht, das erzeugt werden soll. Außerdem kann es sich beispielsweise um natürliche Sprache handeln, die Objekt beschreibt, das der Generator erzeugen soll. Der Diskriminator überprüft dann nicht nur, ob das Bild echt oder künstlich ist, sondern auch, ob es der Bedingung x entspricht.

Die Kostenfunktion lautet dann wie folgt: [25]

$$\min_G \max_D V(D, G) = \mathbb{E}_{x,y}[\log(D(x, y))] + \mathbb{E}_{x,z}[\log(1 - D(x, (G(x, z))))] \quad (2.18)$$

Wobei:

x : Bedingung, die der Generator enthält beziehungsweise zu dem Bild y gehört

y : Reales Bild aus den Trainingsdaten

Bei der Generierung von Straßenschildern könnte das x zum Beispiel eine Zahl sein, die für das Straßenschild steht, das der Generator erzeugen soll.

Zusätzlich ist es möglich, solche cGANs zur Bild-zu-Bild Übersetzung zu verwenden. Darauf bezieht sich gezielt eine bestimmte Veröffentlichung [25]. Dabei sorgt der Vektor z dafür, dass die Ausgabe des Generators nicht-deterministisch ist. Die Autoren der Veröffentlichung schreiben jedoch, dass der Generator das z meistens weitgehend ignoriert und dadurch wenig Varianz in den generierten Bildern zeigt. Der Quellcode der Veröffentlichung trägt den Namen *pix2pix*. [25]

Innerhalb von *pix2pix* basiert der Generator auf einem *U-Net*. Dies ist eine Architektur aus dem Jahre 2015, die das Eingangsbild zunächst auf eine geringere Anzahl an Neuronen Kodiert und anschließend zu einem neuen Bild dekodiert. Die Architektur ähnelt somit konzeptionell der eines Autoencoders. Diese Architektur ist ursprünglich für die Segmentierung, also das markieren von bestimmten Bereichen in Bildern, für medizinische Zwecke entwickelt worden. [26] [25]

Der Diskriminator verwendet eine Architektur, die sich *PatchGAN* nennt. Die Besonderheit ist hierbei, dass der Diskriminator nicht einzelne Pixel betrachtet, sondern Teile eines Bilds als echt oder unecht klassifizieren kann. Die gleiche Diskriminatorarchitektur wird auch in der Veröffentlichung zum CycleGANs verwendet. [25] [21]

Einer der Hauptunterschiede zwischen *pix2pix* und CycleGANs ist, dass *pix2pix* gepaarte Trainingsdaten benötigt. Zu jedem x müssen die Trainingsdaten das erwartete y enthalten. Nur so kann der Diskriminator lernen, welche Bilder zu einer bestimmten Bedingung gehören. CycleGANs brauchen das nicht. Sie benötigen Trainingsbilder aus der Domäne X und aus der Domäne Y, diese müssen jedoch in keiner Beziehung zueinander stehen. Deshalb fallen cGANs unter das überwachte Lernen, während CycleGANs dem unüberwachten Lernen zuzuordnen sind. [25] [21]

Auf Taxonomie eingehen

2.4 Vorherige Arbeiten

Durch eine Recherche haben sich zwei Arbeiten gezeigt, die sich, analog zu dieser Studienarbeit, mit der künstlichen Generierung von Straßenschildern mittels KNNs beschäftigen. Beide

Arbeiten konzentrieren sich darauf, Bildausschnitte zu erzeugen, die ein Straßenschild zeigen und eine geringfügige Menge an Hintergrund um das Schild.

2.4.1 Generierung Taiwanischer Straßenschilder mittels DCGAN

Eine der beiden Arbeiten wurde im Jahr 2021 veröffentlicht. Sie konzentriert sich auf die Generierung taiwanischer Straßenschilder. Dafür wird ein *DCGAN* verwendet, ein GAN, das im Generator und im Discriminator eine tiefe CNN Architektur besitzt. Getestet wird, inwiefern künstlich generierte Trainingsbilder die Erkennung von Straßenschildern verbessern können. Es werden in der Arbeit vier Arten von Verkehrsschildern generiert. [27]

Für jede der vier Klassen wird das GAN mit 350 Bildern trainiert. Die Bildgrößen variieren dabei, wobei die Maximalgröße bei 200x200 Pixel liegt. Die generierten Bildgrößen korrespondieren zu denen der Trainingsbilder. Es sollen in der Arbeit bewusst keine größeren Bilder als 200x200 Pixel erzeugt werden, da Straßenschilder häufig nur einen kleinen Teil des Sichtfelds auf der Straße ausmachen. Das Training erstreckt sich auf bis zu 2000 Epochen, was bedeutet, dass das GAN während des Trainings 2000 mal alle Trainingsbilder als Eingabe erhält. [27]

Da die Anzahl an Trainingsbildern beschränkt ist, generiert das Modell keine völlig neuartigen Bilder, sondern für jede Klasse jeweils vergleichsweise ähnlich aussehende:



Abbildung 2.12: Beispielergebnisse der Generierung taiwanischer Schilder [27]

Zur Beurteilung der Generierung wird mitunter der sogenannte Index struktureller Ähnlichkeit (engl.: Structural Similarity Index) (SSIM) verwendet. Statt dass beispielsweise die Differenz aller entsprechenden Pixelwerte berechnet wird, werden hier die Aspekte *Kontrast*, *Leuchtdichte* und *Struktur* der generierten und der echten Bilder verglichen. Dafür werden keine Berechnungen mit einzelnen Pixelwerten durchgeführt, sondern es wird mit den Mittelwerten und der Standardabweichung der Pixelwerte gerechnet. Auf die zugehörigen Formeln wird in Kapitel 6 eingegangen. [27]

Der Nutzen der generierten Bildern wird anhand eines Modells zur Objektdetektion getestet. In dem Fall, ein sogenanntes *YOLO* Modell. Die Detektion erfolgt auf größeren Bildern, auf denen

mehrere Straßenschilder zu sehen sind. Für das Training des Modells werden mit etwa gleicher Gewichtung die für das GAN verwendete Trainingsdaten und generierte Daten des GANs verwendet. Zur Evaluation wurden hierbei Bilder verwendet, die insgesamt 40 Straßenschilder beinhaltet. Die Ergebnisse können folgender Tabelle entnommen werden: [27]

Modell	Reale Trainingsbilder?	Künstliche Trainingsbilder?	Genauigkeit
Densenet	Ja	Ja	92%
Resnet	Ja	Ja	91%
Densenet	Ja	Nein	88%
Resnet	Ja	Nein	63%

Tabelle 2.1: Vergleich der Objekterkennung mit und ohne künstliche Trainingsdaten

Das Resultat ist, dass die Erkennung durch die generierten Trainingsdaten verbessert wird. Sowohl das Densenet als auch das Resnet liefern durch sie genauere Ergebnisse. [27]

2.4.2 Generierung Deutscher Straßenschilder mittels CycleGAN

Eine weitere Publikation, die sich mit der künstlichen Generierung von Bildern mit Straßenschildern konzentriert, verwendet einen Datensatz, der deutsche Straßenschilder enthält. Er ist unter dem Namen GTSRB bekannt. Auf den Datensatz wird näher in Kapitel 3 eingegangen, da er auch die Basis für diese Arbeit bildet. Die genannte Publikation ist aus einer Masterarbeit an der Ruhr Universität Bochum entstanden. Dort wurde auch der GTSRB veröffentlicht.

Die Architektur des Modells ist hier eine andere als bei der bisher beschriebenen Generierung taiwanischer Schilder. In dieser Arbeit wird ein CycleGAN statt eines *Vanilla GANs* verwendet. Die beiden Generatoren basieren auf einem Resnet. [5] [28]

Für die Generierung erhält das Netzwerk das Piktogramm eines Straßenschildes als Eingang. Das CycleGAN soll einen möglichst realistisch wirkenden Hintergrund um das Straßenschild erzeugen. Vor der Generierung werden die Piktogramme der Straßenschilder zufällig rotiert und ein zufälliger einfarbiger Hintergrund erzeugt. Letzteres soll eine weitere stochastische Komponente für die Generierung bilden, damit eine größere Varianz an Hintergründen erzeugt wird. Für das Training des Netzes verwendet die Arbeit eine präparierte Version des GTSRB. Der Datensatz beinhaltet dadurch folgende Eigenschaften:

- Nur Bilder mit einer Mindestauflösung von 64x64 Pixeln werden verwendet

- Die Bilder werden so zugeschnitten, dass sie quadratisch sind
- Die Klassen werden ausbalanciert, um der asymmetrischen Verteilung an Trainingsbildern pro Klasse entgegenzuwirken
- Insgesamt besteht der präparierte Datensatz aus 12.212 Bildern

[28]

generierte Beispiele einfügen

Es erfolgt eine Evaluation, inwiefern die künstlich generierten Trainingsbilder zwei verschiedene Klassifikatoren verbessern können. Dabei handelt es sich um eine sogenannte Support Vector Machine (SVM) und ein CNN. SVMs sind eine Art von trainierbaren Klassifikatoren, die nicht auf KNNs basieren [9]. Einerseits werden die Algorithmen mit realen Trainingsdaten trainiert und andererseits vollständig mit generierten Daten. Die Ergebnisse bezüglich der Genauigkeit der Klassifikation je nach der Art der Trainingsbilder ist in Tabelle 2.2 dargestellt. Es lässt sich erkennen, dass die Klassifikation um jeweils etwa 7-9% ungenauer ist, als mit realen Trainingsdaten. Es ist jedoch auch erwartbar, dass die Genauigkeit etwas geringer ausfällt, da die generierten Bilder der Verteilung des echten Trainingsdatensatzes folgen sollen, dies jedoch nicht zu 100% möglich ist. [28]

Modell	Reale Trainingsbilder?	Künstliche Trainingsbilder?	Genauigkeit
CNN	Ja	Nein	95,42%
SVM	Ja	Nein	87,97%
CNN	Nein	Ja	87,57%
SVM	Nein	Ja	79,27%

Tabelle 2.2: Vergleich der Klassifikation mit echten und künstlichen Trainingsdaten [28]

2.5 Machine Learning Frameworks

Es ist möglich, KNNs von Grund auf zu programmieren. Die Hauptaufgabe von Entwickler*innen besteht jedoch darin, sowohl den Datensatz als auch die Architektur sowie die Hyperparameter des Modells zu entwerfen und anzupassen. Verschiedene *Frameworks* bieten für die Berechnung der Vorhersagen und das Optimieren eines KNN bereits Funktionen an. Sie sorgen zudem dafür, dass diese Funktionen möglichst performant sind. Hierfür spielen vor allem Grafikkarten (GPUs) eine bedeutende Rolle. Neben sogenannten Tensor Processing

Units (TPUs) und Field Programmable Arrays (FPGAs) werden in erster Linie GPUs für das Berechnen von KNNs eingesetzt. [29]

Einige Frameworks im Bereich des maschinellen Lernens unterstützen eine Berechnung auf GPUs. Dazu zählen unter anderem **TensorFlow**, **PyTorch**, **MXNet**, **Microsoft CNTK** und **Caffe**. Eine Veröffentlichung aus dem Jahre 2019 vergleicht dabei mitunter diese Frameworks. Das am meisten verbreitete Framework sei dabei TensorFlow. Es wurde im Jahre 2015 von der Firma Google entwickelt und ist, wie die meisten anderen genannten Frameworks, überwiegend in der Programmiersprache C++ geschrieben. PyTorch stammt von der Firma Facebook und basiert auf dem Framework *Torch*. Einige Frameworks wie Caffe sind für spezielle Anwendungsgebiete gedacht, wohingegen beispielsweise TensorFlow und PyTorch allgemein Anwendung finden. Anwendende können die Funktionen aller genannten Frameworks mit der Sprache Python nutzen, welche als die für das maschinelle Lernen am meisten eingesetzte Programmiersprache gilt. [29]

Eine bestimmte Art und Weise, wie Frameworks KNNs optimieren ist im Verlauf dieser Arbeit relevant: Viele der genannten Frameworks unterteilen Datensätze in sogenannte *Batches*. Jeder Batch beinhaltet einen Teil des Datensatzes. Eine *Batch Größe* (engl.: *batch size*) legt fest, wie viele Elemente sich in einem Batch befinden. Besitzt der Datensatz 1024 Bilder und das KNN eine Batch Größe von 16, dann wird der Datensatz in 64 Batches unterteilt, da $\frac{1024}{16} = 64$. Es ist möglich, alle Elemente eines Batches gleichzeitig in ein neuronales zu speisen. Bei einer Batch Größe von 16 erhält das KNN pro Trainingsschritt 16 Eingaben und trifft somit eben so viele Vorhersagen. In einem Trainingschritt wird das KNN dann auf allen 16 Vorhersagen trainiert. Die Frameworks sorgen dafür, dass die Batches dynamisch in den Arbeitsspeicher geladen werden. Somit muss der Arbeitsspeicher keine Kapazität für den gesamten Datensatz besitzen. [30]

Weiterhin ist mitunter in TensorFlow und PyTorch der Begriff des *Tensors* relevant. Für diese Arbeit kann ein Tensor als ein mehrdimensionales Array betrachtet werden. Tensoren werden mit einer Stufe beschrieben, die ihre Dimensionalität angibt. Ein Skalar hat die Stufe 0, ein Vektor die Stufe 1 und eine Matrix die Stufe 2. Weiterhin besitzen Tensoren eine Form. Ein Tensor dritter Stufe der Form (256, 256, 3) besitzt eine Höhe und Breite von 256 sowie eine Tiefe von 3. Das kann zum Beispiel ein digitales Bild mit den Pixelmaßen 256x256 und drei Farbkanälen sein. Betrachtet man einen Batch solcher Bilder mit einer Batch Größe von 16, dann erhält man einen Tensor der Stufe vier mit der Form (16, 256, 256, 3). Vorstellen kann man sich das als 16 übereinander gestapelte Bilder. Würde man nun zwei solcher Batches in einem Tensor zusammenfassen, dann hätte dieser Tensor die Form (2, 16, 256, 256, 3) [31].

3 | Konzeption des Modells

3.1 Datensatz

Analog zu der in Kapitel 2.4.2 beschriebenen Arbeit verwendet diese Studienarbeit den GTSRB als Datensatz. Dass dies der größte veröffentlichte Datensatz für deutsche Straßenschilder ist, stellt hierbei den ausschlaggebenden Punkt dar.

Die Bilder des GTSRB verteilen sich auf 43 Klassen respektive 43 verschiedene Arten von Straßenschildern. Eine Auflistung aller Klassen ist im Anhang in Abbildung A.1 dargestellt. Beispielbilder aus dem GTSRB zeigt Abbildung 3.1: [5]



Abbildung 3.1: Beispielbilder aus dem GTSRB Datensatz [5]

Der GTSRB setzt sich aus Bildern zusammen, die unterschiedliche Seitenverhältnisse und verschiedene Auflösungen besitzen. Ein Großteil davon ist kleiner als 100x100 Pixel. Auf jedem Bild ist genau ein Straßenschild zu sehen. Die Bilder basieren auf Videos, die durch die Autoren tagsüber im Straßenverkehr aufgenommen wurden. Dabei sind die Trainingsbilder ungleich auf die Anzahl an Klassen verteilt. Dies hängt mitunter damit zusammen, dass die jeweiligen Schilder nicht gleich häufig im Straßenverkehr vorkommen. Zusätzlich zu den Trainingsbildern besitzt der GTSRB 12.630 Testbilder, welche in dieser Arbeit zur Evaluation des Modells verwendet werden können. Eine nennenswerte Eigenschaft des GTSRB ist, dass eine signifikante Anzahl an Bildern einer Klasse sich ähnlich sehen. Das hängt damit zusammen, dass sie mit zeitlicher Verzögerung aus der selben Fahrsituation stammen. [5]

Die Bilder, die durch diese Studienarbeit generiert werden sollen, haben eine Auflösung von 256x256 Pixel. Auf den genauen Hintergrund hierzu wird zu einem späteren Zeitpunkt eingegangen. Unter anderem deshalb ist der GTSRB für diese Arbeit zunächst so präpariert, dass nur Bilder verwendet werden, die mindestens 50 Pixel breit oder hoch sind. Dies wird im Verlauf geändert, sodass die Mindestgröße 75 Pixel beträgt. Das verringert die Anzahl an verfügbaren

Trainingsbildern signifikant verringert. Der präparierte Datensatz besteht aus 4.510 Bildern, wodurch nur etwa 11% des GTSRB genutzt werden.

Die Verteilung der Daten ist auch im präparierten Datensatz nicht homogen. Das nachfolgende Diagramm zeigt hierfür die Anzahl an Trainingsbildern pro Klasse.

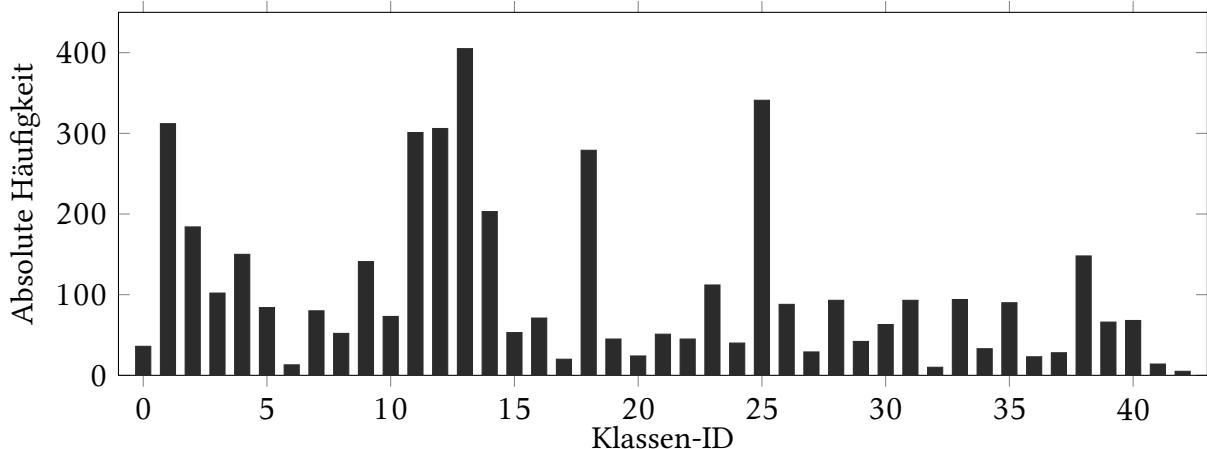


Abbildung 3.2: Häufigkeitsverteilung der Klassen von Straßenschildern im präparierten GTSRB

Eine ungleiche Verteilung der Trainingsdaten kann die Qualität der generierten Bilder negativ beeinflussen. Die in Kapitel 2.4.2 beschriebene Veröffentlichung gibt hierauf bereits Hinweise [28]. Aus diesem Grund, und da die Größe des präparierten Datensatzes als zu gering bewertet werden kann, ist der Datensatz derart erweitert, dass er einerseits mehr Trainingsbilder enthält und andererseits eine gleichmäßige Verteilung der Klassen vorliegt. Dabei wird nicht darauf geachtet, jede einzelne Klasse möglichst gleich oft zu repräsentieren, sondern jede Kategorie von Klassen. Die 43 Klassen werden dazu in die Kategorien *Geschwindigkeitsbegrenzungen*, *Richtungsweiser*, *Aufhebungen*, *Verbotszeichen*, *Gefahrzeichen* und *Einzigartig* unterteilt. Es zeigt sich nämlich, dass das Modell zwischen Straßenschildern, die eine ähnliche Bedeutung und damit auch äußerliche Ähnlichkeiten besitzen, recht gut transferieren kann. Diese Einteilung der Schilder in unterschiedliche Kategorien erfolgt auch in der ursprünglichen Veröffentlichung des GTSRB. Die Kategorisierung für diese Studienarbeit ist identisch, verwendet jedoch deutschen Bezeichnungen für die Kategorien. Im Anhang befindet sich eine Übersicht über die Kategorien von Schildern und ihre zugeordneten Klassen.

Der Großteil an hinzugefügten Trainingsdaten stammt aus der chinesischen *Traffic Sign Recognition Database*, die ein Teil der *Chinese Traffic Sign Database* ist. Dieser Datensatz ist bedeutend kleiner als der GTSRB, bietet jedoch auch einige Bilder mit einer höheren Auflösung als der GTSRB. Somit ist hier ein größerer Anteil des Datensatzes nutzbar. Allgemein ähneln diese Bildern denen des GTSRB. Mit dem Unterschied, dass sie chinesische Straßenschilder zeigen. Für den präparierten Datensatz werden nur die Bilder verwendet, die in eine der genannten Kategorien fallen. Nachfolgend sind Beispiele aus dem Datensatz dargestellt: [32]



Abbildung 3.3: Beispielbilder aus der chinesischen Traffic Sign Recognition Database [32]

Zu sehen ist in Abbildung 3.3 eine Geschwindigkeitsbegrenzung, ein Richtungsweiser, ein Verbotszeichen und ein Schild der Kategorie *Einzigartig*. Der präparierte Datensatz beinhaltet auch Schilder, die nicht durch das Modell dieser Studienarbeit generiert werden sollen. Wie etwa die in Abbildung 3.3 vorhandene Geschwindigkeitsbegrenzung von $15 \frac{km}{h}$. Die Idee ist, dass das Modell diese Bilder dennoch nutzen kann, um die Generierung von anderen Geschwindigkeitsbegrenzungen zu optimieren. Es sind allgemein Unterschiede zu deutschen Straßenschildern vorhanden, die jedoch in dieser Arbeit als vernachlässigbar angenommen werden. Zumindest dann, wenn deutsche Straßenschilder weiterhin den größten Teil des präparierten Datensatzes ausmachen. Eine gewisse Ähnlichkeit ist vorhanden, auch da das Aussehen von Straßenschildern durch das Wiener Übereinkommen über Straßenverkehrszeichen in vielen Ländern weltweit vereinheitlicht ist [33]. [32]

Zusätzlich setzt sich der präparierte Datensatz aus Bildern weiterer Datensätze zusammen. Hier ist jedoch die Anzahl an Bildern signifikant geringer als die der chinesischen Traffic Sign Recognition Database. Zwei der Datensätze bestehen aus Bildern, die eine vollständige Sicht außerhalb des Fahrzeugs zeigen. Hier sind demnach gegebenenfalls mehrere Straßenschilder pro Bild zu sehen, wobei zusätzlich andere Fahrzeuge, Gebäude, Personen und weitere Objekte sichtbar sind. Die Datensätze nennen sich *Mapillary Traffic Sign Dataset* und *BelgianTS Dataset* [34] [35].



Abbildung 3.4: Beispielbild aus dem *Mapillary* Datensatz [34]

Da diese Bilder manuell so zugeschnitten werden müssen, dass sie ähnlich zu dem GTSRB und der chinesischen Traffic Sign Recognition Database einzelne Schilder mit wenig Hintergrund

zeigen, macht diese Menge an Bildern einen geringeren Anteil aus. Aus einem dritten Datensatz werden weniger als 50 Bilder verwendet [36]. Hier sind größtenteils Stoppschildern zu sehen mit einer Bildauflösung von etwa 190 bis zu 300 Pixel in der Höhe oder Breite.

Der vollständige, präparierte Datensatz ist öffentlich unter [diesem Link](#) verfügbar (Stand: 27.03.2023). Er besteht aus 5.809 Bildern. In der nachfolgenden Abbildung ist die Häufigkeitsverteilung der Trainingsdaten (x-Achse) je Straßenschild-Kategorie (y-Achse) zu sehen. Die drei letztgenannten Datensätze sind in der Farbkodierung der Kategorie *Sonstige* zugeordnet.

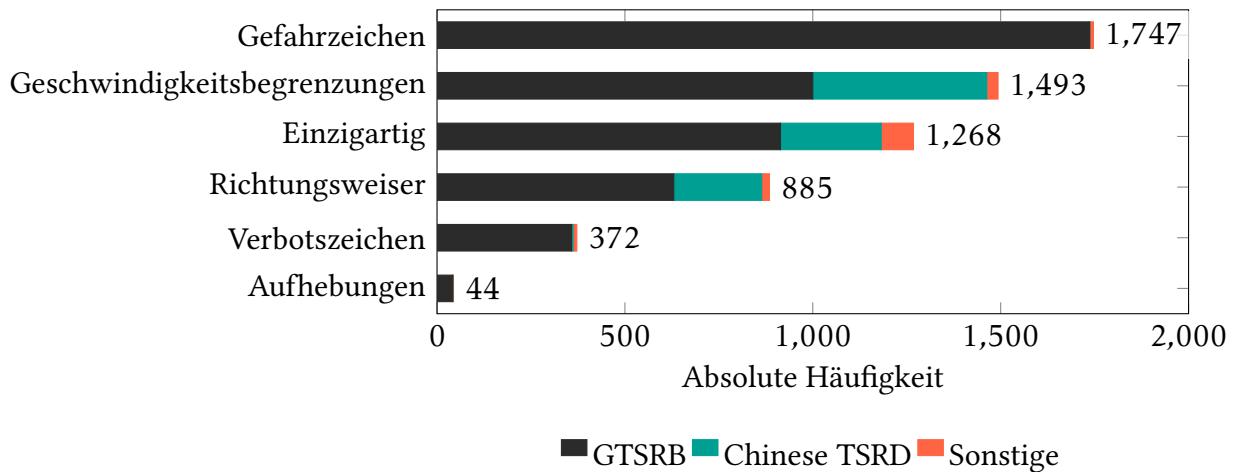


Abbildung 3.5: Häufigkeitsverteilung der Kategorien von Straßenschildern im präparierten Datensatz

Die Kategorie *Aufhebungen* ist nach wie vor signifikant unterrepräsentiert. Eine mögliche Lösung, für die der zeitliche Rahmen nicht ausreicht, wäre, manuell auf [Mapillary](#) nach solchen Bildern zu suchen. Mapillary ist eine Webseite, auf der Nutzer Bilder zu bestimmten GPS-Koordinaten hochladen können. Das Ziel von Mapillary ist, Straßenansichten für alle Straßen auf der Welt bereitzustellen. Hier ist es möglich, explizit nach bestimmten Arten von Straßenschildern zu suchen. Einer der genannten Datensätze stammt aus einer Mischung von weltweiten Mapillary-Bildern. [34]

3.2 Framework

Für das Modell ist nicht nur relevant, auf welchem Datensatz es trainiert wird. Die Wahl eines Frameworks hat einen signifikanten Einfluss die Art der Implementierung. Die Wahl liegt darauf, **TensorFlow** für die Umsetzung zu nutzen. Das hängt einerseits damit zusammen, dass es als das am meisten verbreitete Framework für maschinelles Lernen gilt [29]. Somit existieren hier einige vorgefertigte Implementierungen, die als Basis für die Studienarbeit genutzt werden können. Erwähnenswert ist, dass das bei anderen Frameworks wie etwa PyTorch jedoch ebenso der Fall ist. Ein weiterer Vorteil von TensorFlow ist, dass es standardmäßig von [Google Colab](#)

unterstützt wird. Bei Google Colab ist es möglich, Rechner von Google zu benutzen, um Modelle zu trainieren. Außerdem ist TensorFlow nicht nur in der Forschung, sondern auch in der Industrie verbreitet. Diese Studienarbeit soll ein Verfahren darlegen, das in der Art auch in der Industrie eingesetzt werden könnte. Somit soll hier eine Basis genutzt werden, die dort bereits Verwendung findet. Es wäre jedoch ebenso möglich, ein anderes Framework zu nutzen um vergleichbare Ergebnisse zu erzielen.

Ein Nachteil, den die Literatur bei TensorFlow nennt, ist auf folgendes zurückzuführen: TensorFlow 1.0 übersetzt den Quellcode in Graphen. Das verbessert die Performanz der Berechnungen, sorgt aber dafür, dass die Berechnungen während der Laufzeit statisch sind. Außerdem benötigen einige Operationen dadurch eine spezielle Syntax. Auch erschwert das eine Anbindung anderer Python-Bibliotheken. TensorFlow 2.0, eine neuere Version des Frameworks, erlaubt jedoch zusätzlich eine *Eager Execution*. Hierbei erstellt das Framework standardmäßig keine Graphen. Das Ziel von TensorFlow 2.0 ist unter anderem, einfachere Programmierschnittstellen zu bieten und somit den genannten Nachteil zu beheben. [31]

TensorFlow ist analog zu anderen Frameworks für das maschinelle Lernen auf eine performante Ausführung ausgelegt. Die Geschwindigkeit der Berechnungen hat Auswirkungen auf die Trainingsdauer des Modells. Aus diesem Grund sollen möglichst viele Funktionen in dieser Studienarbeit ausschließlich mit TensorFlow implementiert werden. Andere Python-Pakete werden dann eingesetzt, wenn TensorFlow zu einem bestimmten Problem keine Funktion bietet. Zur Bildverarbeitung stellt TensorFlow die Bibliothek **TensorFlow Graphics** zur Verfügung [37]. Hiermit können nicht nur einzelne Bilder bearbeitet werden, sondern gesamte Batches von Bildern. Die Bibliothek **TensorFlow Addons** bietet zudem zusätzliche Funktionen an, die standardmäßig nicht in TensorFlow vorhanden sind.

In Fällen, in denen TensorFlow Graphics keine geeigneten Funktionen besitzt, wird **OpenCV** oder **Pillow** verwendet. Beides sind Python-Pakete zur Bildverarbeitung, wobei erstere vor allem für Computer-Vision Aufgaben gedacht ist, während Pillow das Manipulieren von Bilddateien erlaubt. [38] [39]

Numpy erklären

...Eine Konvention bei TensorFlow ist, es folgendermaßen zu importieren: `import tensorflow as tf`. Rufen Codeblöcke in dieser Studienarbeit TensorFlow-Funktionen auf, werden sie deshalb als `tf.some_function()` referenziert. ...

3.3 Architektur

In Kapitel 2.3 sind verschiedene generative Netzwerkarchitekturen vorgestellt. Jede dieser Architekturen besitzt ihre Vor- und Nachteile. Die Entscheidung liegt darauf, kein PixelRNN zu benutzen, da die ausschließlich sequentielle Generierung die Performanz des Modells beeinträchtigt. GANs gelten von den vorgestellten Modellen als die am schwierigsten zu trainierende Kategorie. Das hängt vor allem damit zusammen, dass die Kostenfunktion nicht gegen *null* streben soll, sondern gegen einen Wert größer als *null* konvergieren soll. Das Training gilt als instabil, da die Kostenfunktion oszillieren kann und in dem Fall nicht konvergiert. Weiterhin ist es möglich, dass der Generator oder der Diskriminatator jeweils den Gegenspieler soweit überlistet, dass das Modell nicht mehr lernt. Da die Literatur GANs nachgesagt, qualitativ hochwertigere Bilder zu erzeugen als Variational Autoencoder, sollen trotzdem GANs die Basis für diese Studienarbeit bilden. [17] [40]

Die Problemstellung dieser Arbeit soll jedoch nicht als eine reine Bildgenerierung interpretiert werden, sondern als eine Bild-zu-Bild Übersetzung. Damit muss das Netzwerk nicht von alleine lernen, die Symbole der Straßenschilder zu erzeugen. Das ist aus einem bestimmten Grund von Vorteil: Es heißt in der Literatur, dass GANs nicht sonderlich gut darin seien, bestimmte Formen exakt zu erzeugen [17]. Eine reine Bildgenerierung mit einem zufälligen Eingangsvektor könnte also dazu führen, dass das GAN verschwommene oder verformte Schilder erzeugt. Außerdem soll das Netzwerk lernen, verschiedene Arten von Straßenschildern zu erzeugen. Am besten so, dass Anwendende die Arten der generierten Schilder selbst bestimmen können. Das wäre mit einem klassischen cGAN möglich. Da jedoch Straßenschilder genormt sind und somit die Schilder eines Landes stets identisch aussehen, ist es auch möglich, dem GANs das zu erzeugende Schild bereits in minimaler Form als Eingangsbild zu geben. Das Modell muss dann lernen, dieses Bild in die Zieldomäne Y zu übersetzen, die das Schild in einer möglichst fotorealistischen Umgebung zeigt. Was in dem Fall variabel ist, ist die Perspektive des Schildes, die Helligkeit des Bilds sowie das Aussehen der Umgebung. Hier soll das Modell eine möglichst Große Varianz erzeugen.

Die Basis für die Bild-zu-Bild Übersetzung bilden Piktogramme von Straßenschildern. Entnommen sind diese Piktogramme von der Internetseite der *Bundesanstalt für Straßenwesen*. Sie zeigen das jeweilige Straßenschild-Symbol auf einem hellgrauen Hintergrund. Daraus wird ein zusätzlicher Datensatz an Bildern erstellt, der die Domäne X darstellt. Er beinhaltet die Piktogramme für alle 43 Klassen von Straßenschildern, die in dem GTSRB vorkommen. Im Anhang befindet sich eine Abbildung, die alle Piktogramme zeigt. Abbildung 3.6 soll verdeutlichen, was die Domänen X und Y sind, die das Modell ineinander übersetzen soll. [41]

Eine letzte Fragestellung ist, ob cGANs analog zu pix2pix verwendet werden sollen oder aber CycleGANs. Wie bereits beschrieben, benötigt pix2pix gepaarte Trainingsdaten. Zu jedem



Abbildung 3.6: Domänen für die Bild-zu-Bild Übersetzung

echten Bild aus Y muss hinterlegt werden, welches Piktogramm aus X dazu gehört. Das kann insofern als unproblematisch betrachtet werden, als dass die Bilder des GTSRB nach ihren Klassen sortiert sind. Soll diese Studienarbeit jedoch mit einem größeren Datensatz fortgeführt werden, ist es von Vorteil, wenn die Trainingsdaten nicht annotiert werden müssen. Außerdem schreiben die Autoren von pix2pix, dass die erzeugten Bilder keine sonderlich hohe Varianz aufzeigen würden, da, wie bereits erwähnt, der Vektor z einen geringen Einfluss auf die Generierung habe. Die Veröffentlichung der CycleGANs baut auf pix2pix auf. Es wird davon ausgegangen, dass die Bildgenerierung deshalb nicht signifikant schlechter, oder noch besser ist als mit pix2pix, während das Modell die genannten Vorteile besitzt. Aus diesem Grund basiert das Modell für diese Studienarbeit auf einem CycleGANs.

3.4 Datenaugmentation

Bevor die Piktogramme an den Generator übergeben werden, werden sie zufällig rotiert. Dadurch muss der Generator die Rotation nicht eigenständig lernen und dieser Aspekt der Generierung lässt sich deterministisch bestimmen. Dabei soll die Rotation nicht nur in x-y-Richtung erfolgen, sondern auch eine dreidimensionale Rotation simuliert werden. Und zwar so, als sei das Schild aus einer beliebigen Frontalperspektive aufgenommen worden.

Um bestimmte Transformationen eines Bilds mittels einer Matrixmultiplikation darstellen zu können, wird häufig ein sogenanntes *homogenes Koordinatensystem* verwendet. Dabei wird das Koordinatensystem um eine weitere Dimension erweitert. Ein Punkt $p = [x, y]^T$ kann somit um einen beliebigen Wert in z-Richtung verschoben werden. Dadurch wird ein Punkt \tilde{p} im homogenen Koordinatensystem durch drei Koordinaten \tilde{x} , \tilde{y} und \tilde{z} beschrieben. Transformationen werden in der homogenen Darstellung durchgeführt und anschließend werden daraus die kartesischen Koordinaten x und y bestimmt. Somit erhält man aus der Transformation erneut ein zweidimensionales Bild. [42] [43]

Dies wird für eine dreidimensionale Rotation der Piktogramme benötigt. Die Rotation soll durch drei *eulersche Winkel* beschrieben werden. Das bedeutet, dass sie sich aus einer Rotation um die z-Achse, einer um die y-Achse und einer um die x-Achse zusammensetzt. Dies ist in

Abbildung 3.7 gezeigt. Die bläulichen Balken zeigen dabei die Achse an, um die gedreht wird. Die erste Rotation ist um die z-Achse, wodurch der Balken in die dritte Bildebene geht. [43]



Abbildung 3.7: Rotation der Straßenschilder mittels eulerscher Winkel

Jede Rotation ist durch einen einzelnen Winkel um die jeweilige Achse bestimmt. Kombiniert man die Rotationen, kann die resultierende Transformation somit durch drei Winkel ($\alpha_z, \alpha_y, \alpha_x$) eindeutig beschrieben werden. Für die Erzeugung einer zufälligen Rotation müssen randomisierte Werte für diese Winkel bestimmt werden. [43]

Zusätzlich zu der Rotation, soll das Modell die Piktogramme zufällig in ihrer Größe skalieren. Die genannten Augmentationen dienen dazu, die Verteilung der real aufgenommenen Schilder abbilden zu können. Im Datensatz besitzen die Schilder eine unterschiedliche Größe und sind aus verschiedenen Perspektiven aufgenommen. Dadurch dass die Augmentation deterministisch ist, kann sie dazu genutzt werden, um gezielt nur Bilder durch das Modell zu generieren, die aus bestimmten Perspektiven und mit festgelegten Größen generiert wurden. Alternativ kann auch die randomisierte Augmentation beibehalten werden, um eine möglichst große Bandbreite an unterschiedlichen Bildern zu erzeugen.

3.5 Training

Das Training basiert auf den in Kapitel 2.3.3 vorgestellten Verlustfunktionen für CycleGANs. In Abbildung 3.8 sind hierfür die drei Trainingsschritte des CycleGAN dargestellt. Die ersten beiden Schritte berechnen den *Adversarial Loss* von Generator G und Diskriminatoren D_Y , beziehungsweise von Generator F und Diskriminatoren D_X . Was hier trainiert wird, ist die Übersetzung von Domäne X in Y, beziehungsweise von Domäne Y in X. Im Anschluss daran erfolgt die Berechnung des *Cycle Consistency Loss*. Das ist der Trainingsschritt der überprüfen soll, dass die von Generator G erzeugten Bilder das erwartete Straßenschild zeigen. Dazu erzeugt G aus einem Piktogramm das Bild eines Straßenschildes woraus F wiederum das Piktogramm erzeugen soll. Der Generator G ist hierbei grün hervorgehoben, da dies das einzige KNN ist, dass für die praktische Generierung von Bildern verwendet wird. Die weiteren KNNs sollen lediglich den Generator G trainieren. [21]

Für bestimmte Anwendungsfälle schlägt das CycleGAN Paper vor, einen *Identity Loss* hinzuzufügen. Dabei wird Generator G ein echtes Bild eines Straßenschildes und Generator F ein echtes

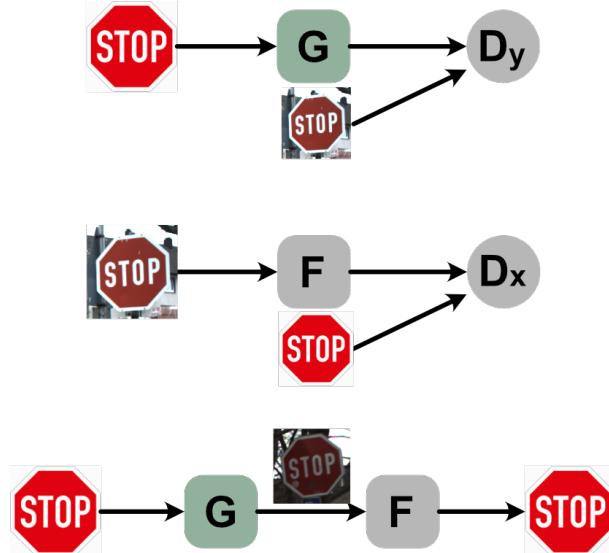


Abbildung 3.8: Trainingsschritte des CycleGAN

Bild eines Piktogramms zugeführt. Da das Eingabebild für G und F bereits aus der Zieldomäne entstammt, wird hier von den Generatoren erwartet, dass sie das Eingabebild möglichst wenig verändern. Die Veröffentlichung schlägt das vor, wenn die Generatoren beispielsweise die Farben der Eingangsbilder beibehalten sollen. [21]

Die Vermutung ist, dass der Identity Loss auch für diese Studienarbeit sinnvoll sein kann. Hierdurch könnte das Netzwerk dazu gebracht werden, das Straßenschild möglichst wenig zu verformen und es könnte aus den Eingabebildern erlernen, verschiedene Hintergründe um die Schilder zu erzeugen. Deshalb wird der Identity Loss in diesem Projekt erprobt. Eine Veröffentlichung deutet außerdem darauf hin, dass der Identity Loss die allgemeine Qualität der generierten Bilder verbessern kann [44]. [21]

Zusätzlich schreiben die Autoren der CycleGAN Veröffentlichung, dass es sinnvoll sein kann, den *Adversarial Loss* mit einer \mathcal{L}_2 Verlustfunktion zu berechnen statt mit einer Binary Cross-entropy Verlustfunktion. Das soll das Training stabilisieren. Die pix2pix-Veröffentlichung erwähnt das ebenfalls [25]. Streng genommen handelt es sich dabei dann bei den Paaren G und D_y , beziehungsweise F und D_x nicht mehr um klassische GANs, sondern um sogenannte *least squared GANs (LS-GANs)*. In dieser Studienarbeit soll das Training zunächst mit einer klassischen CycleGAN Architektur erfolgen. Zeigt sich ein instabiles Training, soll getestet werden, ob sich das Training mittels \mathcal{L}_2 Verlustfunktionen stabilisieren lässt. [21]

Das Training gilt als beendet, wenn die Verlustfunktionen des Modells gegen einen Wert konvergieren. Um das zu messen, ist eine Form des *Loggings* notwendig. Es müssen demnach über den Verlauf des Trainings die Werte der Kostenfunktionen gespeichert werden.

4 | Implementierung und Training

Die Implementierung des CycleGAN basiert auf den bisher beschriebenen Entscheidungen. In einem Ordner `src` befinden sich alle Dateien der Implementierung. Hier existieren neben Dateien für das Training und die Anwendung des Modells auch eine Konfigurationsdatei und verschiedene Hilfsfunktionen. Die Ordnerstruktur der Implementierung ist im Anhang abgebildet. Im wesentlichen geht dieses Kapitel auf folgende Bestandteile ein:

- Die Konfigurationsdatei für das Projekt befindet sich im Pfad `config/config.toml`
- Selbst-definierte Hilfsfunktionen befinden sich im Python-Modul `utils`
- Die Datei `model.py` implementiert das CycleGAN Modell
- Das Modell wird mittels des Python-Skripts `train.py` trainiert
- Das Modell kann mittels des Python-Skripts `generate.py` zum Generieren von Bildern genutzt werden

Das CycleGAN ist als Klasse implementiert und lässt sich somit innerhalb der Skripte instanzieren. Die Konfigurationsdatei ist eine TOML-Datei. Tom's Obvious Minimal Language (TOML) ist eine Konfigurationssprache, die Daten als Schlüssel-Werte-Paare speichert. Die Konfigurationsdatei ist in die Kategorien `paths`, `model` und `training` unterteilt. Anwendende können hier die Pfade zu den Trainingsdaten oder aber Parameter wie `batch_size` oder `number_of_epochs` angeben.

4.1 Modell

Das Modell ist in der Datei `model.py` implementiert. Innerhalb dieser Datei existiert eine Klasse `CycleGan`, die im wesentlichen die in Tabelle 4.1 aufgeführten Methoden besitzt. Der Inhalt dieser Klasse basiert zum Teil auf einem Beispiel von TensorFlow [22]. Dieses Beispiel beinhaltet eine Implementierung eines CycleGAN. Teile des Quellcodes stellen die Basis für die Umsetzung der Studienarbeit dar. In erster Linie basieren die Trainingsfunktionen des CycleGAN auf dem Beispiel sowie der Fakt, dass das Modell die Bilder auf Pixelwerte zwischen -1 und 1 skaliert. Die Skalierung kann das Training des Modells beschleunigen und stabilisieren [45]. Das Modell aus dem TensorFlow Beispiel verwendet einen U-Net Generator, wie es

¹ Vollständiger Methodename: `restore_latest_checkpoint_if_exists`

Methode	Aufgabe
<code>__init__</code>	Initialisieren des CycleGAN mittels der Konfigurationsdatei
<code>generate</code>	Generieren eines einzelnen Batches von Bildern
<code>fit</code>	Trainieren des CycleGANs über mehrere Epochen
<code>train_step</code>	Durchführen eines einzelnen Trainingsschritts
<code>restore_..._checkpoint_...¹</code>	Laden der aktuell gespeicherten Parameter des Modells

Tabelle 4.1: Auswahl an Methoden aus der CycleGAN Klasse

normalerweise bei pix2pix statt bei CycleGANs der Fall ist. Aus diesem Grund beinhaltet die Implementierung dieser Studienarbeit zusätzlich einen ResNet Generator und den dazugehörigen Diskriminator. Diese befinden sich in dem Pfad `src/external/resnet.py` und entstammen ebenfalls einer bereits existierenden CycleGAN Implementierung [23]. Die Architekturen der KNNs entsprechen dabei den Vorgaben der CycleGAN Veröffentlichung [21]. Die Klasse `CycleGAN` erlaubt eine Auswahl zwischen der U-Net- und der ResNet-Architektur. Das Ziel dieser Studienarbeit ist damit unter anderem zu prüfen, ob ein U-Net- oder ein ResNet-basierter Generator für diesen Anwendungsfall besser geeignet ist.

Nachfolgend soll näher auf einige Methoden der Klasse CycleGAN eingegangen werden.

4.1.1 Konstruktor

Die `__init__`-Funktion stellt in Python den Konstruktor einer Klasse dar. Die Klasse `CycleGAN` erwartet hier den Parameter `config`. Dies ist ein Python-Dictionary, das die Werte der Konfigurationsdatei enthält. Es existieren hier somit Schlüssel-Werte-Paare für die verschiedenen Trainingseinstellungen des Modells. Ein Vorteil dieses Ansatzes ist, dass Anwendende des Modells die Parameter in der TOML-Datei ändern können, ohne den Python-Quellcode aufrufen zu müssen. Allgemein verfolgt die Implementierung des CycleGAN das Ziel, dass Anwendende sich mit keinem Python Quellcode auseinander setzen müssen, um das Modell zu trainieren und zu verwenden.

Eines der Attribute der Klasse `CycleGAN` heißt `generator_type`. Es handelt sich dabei um ein Enum, das bestimmt, ob der Generator ein U-Net oder ein ResNet ist. Der Wert für das Attribut ist durch die Konfigurationsdatei und somit durch den Parameter `config` bestimmt. In Listing 4.1 ist der Codeabschnitt zu sehen, der die `generator_type`-Eigenschaft der Klasse `CycleGAN` initialisiert.

```

class GeneratorType(Enum):
    RESNET = 0
    U_NET = 1

class CycleGan:
    ...
    def __init__(self, config):
        ...
        if config['model']['generator_type'] == 'unet':
            self.generator_type = GeneratorType.U_NET
        elif config['model']['generator_type'] == 'resnet':
            self.generator_type = GeneratorType.RESNET
        ...

```

Listing 4.1: model.py - Auswahl der Generator-Architektur

Das Modell verwendet den U-Net Generator mit drei Farbkanälen und einer *Instance Normalization* (hier `'instancenorm'`) statt einer *Batch Normalization*. Die pix2pix-Veröffentlichung schlägt dies für die Bildgenerierung vor [25]. Beides sind bestimmte Schichten in CNNs zur Normalisierung von Werten. Die Klasse `CycleGan` import die Generator- und Diskriminatiorarchitekturen von pix2pix aus dem GitHub Repository von *TensorFlow Examples* als Python-Modul `pix2pix` [46]. Das ist analog zu dem Quellcode des genannten TensorFlow Beispiels [22].

Der ResNet Generator erhält als einen Parameter die Dimensionen des Bilds, das er generieren soll. Hier ist demnach nicht nur die Anzahl an Farbkanälen variabel, sondern auch die Höhe und Breite des Bilds. Der U-Net Generator erzeugt hingegen Bilder mit einer Höhe und Breite von 256 Pixeln. Ein weiterer Parameter nennt sich `n_blocks`. Er gibt die Anzahl an Residual Blocks an. Die CycleGAN-Veröffentlichung schlägt hier für eine Bildgröße von 256x256 einen Wert von 9 vor [21]. Falls dieser Wert zu keinen zufriedenstellenden Ergebnissen führt, soll er verändert werden. Listing 4.2 zeigt die Initialisierung der Generatoren abhängig von der gewählten Generator-Architektur.

```

...
# Generators
if self.generator_type == GeneratorType.U_NET:
    self.generator_g = pix2pix.unet_generator(3,
        ↵ norm_type='instancenorm')
    self.generator_f = pix2pix.unet_generator(3,
        ↵ norm_type='instancenorm')
else:
    self.generator_g = resnet.ResnetGenerator((self.image_size,
        ↵ self.image_size, 3), n_blocks=9)

```

```

self.generator_f = resnet.ResnetGenerator((self.image_size,
    ↳ self.image_size, 3), n_blocks=9)
...

```

Listing 4.2: model.py - Initialisierung der Generatoren

Analog dazu initialisiert die Klasse `CycleGan` die Diskriminatoren D_y und D_x . Die Klasse besitzt weitere Attribute, die für die weiteren Methoden der Klasse von Relevanz sind.

4.1.2 fit-Methode

Bevor das Modell zur Generierung von Bildern genutzt werden kann, muss es trainiert werden. Hierfür existiert die `fit`-Methode. Die vollständige Methode ist aufgrund ihrer Länge im Anhang in Listing 2 abgebildet. Als Parameter benötigt die Methode zum einen den Datensatz an Piktogrammen und zum anderen den Trainingsdatensatz mit realen Straßenschildaufnahmen. Diese Parameter tragen die Bezeichnungen `pictograms` und `real_images`. Die beiden Datensätze müssen dabei als `tf.data.Dataset` Objekte übergeben werden [30]. Die Klasse `tf.data.Dataset` ist Teil des `tf.data` Frameworks. Sie ist explizit auf die Performanz beim Laden großer Datensätze ausgelegt. Über `tf.data.Dataset` Objekte kann beispielsweise mittels einer `for`-Schleife iteriert werden. Bei jeder Iteration gibt der Datensatz dabei einen Batch zurück. Somit muss sich nicht manuell um das Laden einzelner Elemente des Datensatzes gekümmert werden. Auch erfolgt das Laden der Batches asynchron. Batches werden dann in den Arbeitsspeicher geladen, wenn sie benötigt werden. [30]

Ein optionaler Parameter ist zusätzlich die Anzahl an Epochen, die das Modell trainieren soll. Die Anzahl an Epochen ist standardmäßig auf 1 gesetzt. Listing 4.3 zeigt die Funktionsdeklaration sowie einen Ausschnitt aus der Implementierung.

```

def fit(self, pictograms, real_images, epochs=1):
    ...
    for epoch in range(epochs):
        ...
        # Single training step
        for image_batch in tqdm(real_images):
            ...
            # Transform the pictograms
            pictograms.shuffle(buffer_size=100,
                ↳ reshuffle_each_iteration=True)
            single_pictogram_batch =
                ↳ pictograms.take(1).get_single_element()
            single_pictogram_batch, _, _ =
                ↳ utils.preprocess_image.randomly_transform_image_batch(

```

```
    single_pictogram_batch)

# Train the model
losses = self.train_step(single_pictogram_batch,
                         image_batch)
...
```

Listing 4.3: model.py - fit -Methode

Die `fit`-Methode iteriert zunächst über die Anzahl an Trainingsepochen. In jedem Durchlauf trainiert das CycleGAN über den gesamten Trainingsdatensatz. Das realisiert die zweite `for`-Schleife. Hier iteriert die Funktion über das `tf.data.Dataset` Objekt `real_images`. Bei jeder Iteration gibt das Objekt einen neuen Batch an Bildern zurück. Die Funktion `tqdm` stammt aus dem gleichnamigen Python-Paket. Sie dient dazu, eine Fortschrittsanzeige (*engl.: progress bar*) in der Konsole anzuzeigen, die sich nach jeder Epoche aktualisiert.

Bevor das Modell auf einem einzelnen Batch trainiert, muss die `fit`-Methode die Piktogramme zunächst augmentieren. Das besteht aus verschiedenen Schritten. Als erstes wird der Datensatz der Piktogramme gemischt. Dies geschieht mit Hilfe der Methode `shuffle`. Anschließend erzeugen die Methoden `take(1)` und `get_single_element` daraus einen neuen Datensatz, der nur einen Batch aus dem Piktogramm-Datensatz beinhaltet und geben diesen einzigen Batch zurück. Zusammengefasst dient dieser Codeabschnitt dazu, die Piktogramme zufällig zu mischen und einen einzelnen Batch an Piktogrammen daraus zu entnehmen.

Zur Augmentierung der Piktogramme ruft die `fit`-Methode dann die Methode `randomly_transform_image` auf. Diese Methode ist in dem für diese Studienarbeit erstelltem Modul `utils` implementiert. Diese Methode gibt nicht nur die augmentierten Bilder zurück, sondern auch sowohl eine Liste der Rotationsmatrizen als auch der zufälligen Skalierung der Piktogramme. Für das Training werden nur die augmentierten Bilder benötigt. Die zusätzlich zurückgegebenen Daten sind hierfür nicht notwendig. Dass die Rückgabewerte der Methode `randomly_transform_image_batch` nicht benötigt werden, soll der Variablenname `_` signalisieren.

Anschließend dazu führt die Methode `train_step` den eigentlichen Trainingsdurchlauf durch. Sie erhält dabei den zufälligen Batch an Piktogrammen sowie den Batch an realen Straßenschildaufnahmen aus der `for`-Schleife. Damit berechnet sie die Verlustfunktionen des CycleGAN und führt basierend darauf Gradientenabstiege für G , F , D_y und D_x aus. Da der Inhalt dieser Methode größtenteils durch das TensorFlow Beispiel gegeben ist, soll darauf nicht im Detail eingegangen werden. Die Methode implementiert die Kostenfunktionen 2.17 sowie den Identity Loss. Es wird hier jedoch von Verlust- statt von Kostenfunktionen gesprochen, da sie sich hier auf einen einzelnen Batch statt auf den gesamten Datensatz beziehen. Deshalb verwenden die Veröffentlichungen häufig den Begriff *loss* für die Kostenfunktionen. Hat die

`train_step`-Methode die Verlustfunktionen berechnet, führt sie den Gradientenabstieg für das CycleGAN durch.

4.1.3 generate-Methode

Ist ein CycleGAN trainiert, dient die `generate`-Methode zur Generierung von Bildern. Dafür wird lediglich der Generator G benötigt, der Bilder von Piktogrammen in Bilder von Straßenschildern übersetzt. Die `generate`-Methode besteht deshalb nur aus einer Zeile Code. Als Parameter erhält die Methode einen Batch an Piktogrammen, welche sie an Generator G übergibt. Aus diesen Piktogrammen erzeugt der Generator Bilder von Straßenschildern. Die Methode gibt diese Bilder anschließend zurück. Listing 4.4 zeigt die `generate`-Methode.

```
def generate(self, pictograms):
    return self.generator_g(pictograms)
```

Listing 4.4: `model.py` - generate-Methode

4.2 Datenaugmentierung

In Kapitel 3.4 ist das Vorgehen für die Datenaugmentierung beschrieben. Wie bereits gezeigt, nutzt beispielsweise die `fit`-Methode die Datenaugmentierung, um dem Generator G die Größe und Perspektive des Straßenschildes vorzugeben. Hier soll auf die konkrete Implementierung dessen eingegangen werden. Zur Augmentierung müssen die Piktogramme der Straßenschilder zufällig rotiert und skaliert werden. Hierzu dient die Bibliothek Tensorflow Graphics.

Die Augmentierung ist in der Datei `utils/preprocess_image.py` implementiert. Die Funktion, die hierbei aus der CycleGAN Klasse aufgerufen wird, ist `randomly_transform_image_batch`. Das Listing 1 im Anhang zeigt die vollständige Implementierung. Die Funktion erhält einen vierdimensionalen Tensor `img_tensor_batch` als Eingang. Dieser Tensor beinhaltet den Batch an Bildern, der transformiert werden soll. Zunächst skaliert die Funktion zufällig den Inhalt dieser Bilder. Anschließend führt sie darauf eine zufällige dreidimensionale Rotation aus und gibt die transformierten Bilder zurück. Was sie ebenfalls zurückgibt, sind die Listen `content_sizes` und `rotation_matrices`. Die Anzahl an Elementen der Listen entspricht der Größe des übergebenen Batches, also der Anzahl an transformierten Bildern. Hierdurch kann die aufrufende Funktion für jedes Bild identifizieren, welche Zufallswerte für die Transformation generiert wurden. Dies kann genutzt werden, um die Transformation zu replizieren. Genutzt wird das in Kapitel 5, um Bilder von als ungültig markierten Schildern zu erzeugen.

4.2.1 Skalierung

Die Skalierung des Bildinhalts besteht aus mehreren Schritten. Die Funktion `randomly_transform_image_b` generiert zunächst mittels Numpy eine Liste an zufälligen `content_sizes`. Danach skaliert die Funktion die Piktogramme der Straßenschilder auf die in `content_sizes` gespeicherten Pixelgrößen mittels der Hilfsfunktion `resize_content_of_img`. Das ist in Listing 4.5 gezeigt.

```
content_sizes_tmp = content_sizes[:]
transformed_imgs = tf.map_fn(lambda img: resize_content_of_img(img,
    target_size, content_sizes_tmp.pop(0)), img_tensor_batch)
```

Listing 4.5: Skalieren der Bild-Tensoren

Als ersten Schritt dupliziert die Funktion die Liste `content_sizes` in der Variable `content_sizes_tmp`. Anschließend skaliert die Funktion die Bilder. Dazu nutzt sie die von TensorFlow bereitgestellte Funktion `tf.map_fn`. Diese Funktion erhält als Parameter einen Tensor der Stufe n und eine Funktion. Sie führt die Funktion auf jedem Element der Stufe $n - 1$ des Tensors aus. Beispielsweise auf jedem Bild eines Batches von Bildern. In diesem Fall übergeben wir einen Tensor der Stufe vier und der Form (*Batch Größe, Breite, Höhe, Anzahl Farbkanäle*) an die Funktion `tf.map_fn`. Die Funktion erstellt daraus eine Menge an Tensoren der Stufe drei, wobei jeder dieser Tensoren ein Bild mit der Form (*Breite, Höhe, Anzahl Farbkanäle*) ist. Auf jedem Bild führt die Funktion `tf.map_fn` anschließend die Funktion `resize_content_of_img` aus. Letztere Funktion erhält ein Bild und die Zielgröße, auf die der Inhalt des Bilds skaliert werden soll. Anschließend fügt die Funktion `tf.map_fn` die Bilder wieder zu einem einzelnen Tensor der Stufe vier zusammen und gibt ihn zurück. [47]

Es wäre ebenso möglich, über die einzelnen Bild-Tensoren des Batches mittels einer `for`-Schleife zu iterieren. Die Dokumentation von `tf.map_fn` gibt jedoch explizit an, dass sie eine parallele Ausführung ermöglicht. Das ist hier der ausschlaggebende Vorteil gegenüber einer `for`-Schleife. Es ist jedoch dennoch weniger performant als eine Funktion zu verwenden, die eine einzelne Operation vektorisiert auf dem gesamten Tensor ausführt. Warum die Funktionen dennoch mit `tf.map_fn` arbeitet, statt alle Bilder des Batches gleichzeitig zu transformieren, soll der folgende Abschnitt klären. [47]

Die Funktion `resize_content_of_img` verwendet zwei Funktionen aus dem TensorFlow Framework: Die Funktion `tf.image.resize` um das Bild zu skalieren und die Funktion `tf.image.resize_with_crop_or_pad` um das Bild zurück auf die ursprüngliche Größe zu bringen. Wird das Piktogramm verkleinert, dann fügt letztere Funktion einen weißen Rand um das Piktogramm hinzu. Wird es hingegen vergrößert, schneidet die Funktion die Pixel ab, die über die Zielgröße hinausgehen. Obwohl es möglich wäre, den vierdimensionalen Tensor

an beide TensorFlow Funktionen zu übergeben, um alle Bilder gleichzeitig zu skalieren, erhält `resize_content_of_img` lediglich dreidimensionale Tensoren, sprich einzelne Bilder, als Parameter. Das hängt damit zusammen, dass die Piktogramme in einem Batch unterschiedliche Skalierungen besitzen sollen. Die TensorFlow Funktionen sind jedoch nur dazu in der Lage, eine bestimmte Skalierung auf allen Bildern des Batches auszuführen.

Die Rolle der Liste `content_sizes_tmp` ist, dass während jedem Durchlauf der Funktion `tf.map_fn`

4.2.2 Rotation

Der zweite Schritt der Augmentation ist, dass das Modell die Piktogramme rotiert. Es existieren Rotationsmatrizen, die Rotationen mittels eulerscher Winkel beschreiben. Die Drehungen um die x-, y- und z-Achse besitzen jeweils eigene Rotationsmatrizen R_x , R_y und R_z . Der Aufbau dieser Matrizen ist der Literatur entnommen [43]. Um daraus eine einzelne Rotationsmatrix R zu erhalten, werden die Rotationsmatrizen miteinander multipliziert. Dazu dient die Funktion `create_rotation_matrix`. Sie erhält die drei Winkel α_x , α_y und α_z als Parameter und gibt eine einzelne Rotationsmatrix R zurück.

Bevor die Funktion `randomly_transform_image_batch` die Funktion `create_rotation_matrix` aufruft, muss sie ausgehend davon zunächst zufällige Winkel $(\alpha_x, \alpha_y, \alpha_z)$ erzeugen. An dieser Stelle entstammen die zufälligen Winkel jedoch nicht einer Gleichverteilung, sondern einer gaußschen Normalverteilung. Das hängt damit zusammen, dass die Piktogramme in den meisten Fällen nur leicht rotiert sein sollen. Nur ein vergleichsweise geringer Prozentsatz der Piktogramme soll stark Augmentiert sein. Dies soll in etwa nachbilden, aus welchen Perspektiven die Straßenschilder im Trainingsdatensatz aufgenommen sind. Listing 4.6 zeigt die zufällige Generierung der Rotationswinkel α_z :

```
alpha_z_values = np.random.normal(loc=0.0, scale=3.5,
                                   size=batch_size)
```

Listing 4.6: Augmentierung eines Batches von Bildern

Der Parameter `loc` gibt den Erwartungswert der Winkel an, während `scale` die Standardabweichung setzt. Durch das Angeben der `batch_size` wird deutlich, dass die Numpy Funktion hier nicht nur einen Winkel erzeugt wird, sondern ein *Numpy Array* das für jedes Piktogramm einen zufälligen Winkel enthält. Die Implementierung ist somit hier vektorisiert. Im Mittel liegen die Winkel der Rotation bei 0.0° . Der Wert für die Standardabweichung ist empirisch bestimmt, da die Werte für die Winkel nicht den tatsächlichen Winkeln in Grad entsprechen. Die Funktion `randomly_transform_image_batch` erzeugt die Winkel α_y und α_x analog

hierzu, mit dem Unterschied, dass sie dort andere Standardabweichungen (`scale`) als Parameter setzt. Dass eine gaussche Normalverteilung verwendet wird, bedeutet, dass die meisten Piktogramme zu einer Aufnahme aus der Frontalperspektive führen. Der Großteil der Rotationswinkel ist demnach vergleichsweise gering. Einige wenige Schilder hingegen sind stärker rotiert. Würde die Funktion eine Gleichverteilung zur Erzeugung der Winkel verwenden, wäre der Anteil an starken Rotationen in etwa gleich zu dem Anteil an geringen Rotationen.

Die eigentliche Rotation setzt die TensorFlow Graphics Funktion `perspective_transform` um. Sie erhält einen Tensor der Stufe vier an Bildern und einen Tensor der Stufe vier an Rotationsmatrizen. Das bedeutet, dass der gesamte Batch an Bildern samt seiner Rotationsmatrizen übergeben wird. Somit erfolgt die Augmentation der Bilder hier vektorisiert und damit parallel. Die Codezeile `transformed_imgs = 1 - transformed_imgs` kehrt vor der Rotation zunächst die Farbwerte des Bilds um. Das ist nötig, da der Hintergrund, den die genannte TensorFlow Funktion erzeugt, schwarz ist. Kehrt man zunächst die Farbwerte um, führt die Rotation aus und setzt die Farbwerte auf ihren ursprünglichen Wert zurück, so wird der schwarze Hintergrund durch einen weißen ersetzt.

4.3 Training

Anwendende können das CycleGAN mit dem Skript `train.py` trainieren. Prinzipiell besitzt dieses Skript zwei Aufgaben: Es lädt sowohl den Trainingsdatensatz als auch die Piktogramme und ruft die Trainingsfunktion des Modells CycleGAN auf.

4.3.1 Laden der Datensätze

Die Konfigurationsdatei enthält in der Kategorie `paths` den Eintrag `train_data`. Das ist der relative oder absolute Pfad zu dem Trainingsdatensatz. Das Skript `train.py` lädt den Datensatz in ein `tf.data.Dataset` Objekt. Dafür stellt TensorFlow eine Funktion bereit, mit der ein Datensatz an Bildern aus einem Dateipfad geladen werden kann. Anhand der Ordnerstruktur sortiert die Funktion die Bilder automatisch in ihre Klassen ein. Die Funktion nennt sich `load_image_dataset_from_directory` [48]. In folgendem Listing ist der Teil des Skripts dargestellt, der mittels dieser Funktion den Datensatz lädt.

```
training_path = config['paths']['train_data']

x_train = tf.keras.utils.image_dataset_from_directory(training_path,
    batch_size=BATCH_SIZE, image_size=(IMAGE_SIZE, IMAGE_SIZE),
    labels=None, shuffle=True, crop_to_aspect_ratio=True)
```

```
x_train_processed = utils.load_data.normalize_dataset(x_train)
```

Listing 4.7: train.py - Laden des Trainingsdatensatzes

An die genannte TensorFlow Funktion übergibt das Skript mitunter den Pfad zu den Trainingsdaten, die Batch Größe und die Bildauflösung. Die Auflösung muss deshalb übergeben werden, da die Funktion `load_image_dataset_from_directory` alle Bilder auf diese Größe skaliert. Hierzu nutzt die Funktion standardmäßig *bilineare Interpolation*. Dadurch erscheint das Bild nicht als *verpixelt*, sondern fehlende Pixel, die bei der Vergrößerung unweigerlich auftreten, werden durch eine Kombination der benachbarten Pixel aufgefüllt. Dadurch wirkt das Bild statt *verpixelt* eher *verwaschen*. Das CycleGAN benötigt die Daten nicht nach ihren Klassen sortiert, da es mit unüberwachtem Lernen arbeitet. Deshalb wird zusätzlich der Parameter `labels` auf `None` gesetzt.

Durch den nächsten Parameter `shuffle` erfolgt die Einstellung, dass der Datensatz zufällig durchmischt werden soll. Abschließend folgt ein entscheidender Parameter, der einer näherer Erläuterung bedarf. Wie bereits in Kapitel 3.1 beschrieben, besitzen die Trainingsbilder des Datensatzes verschiedene Auflösungen. Das bedeutet, dass Bilder die nicht quadratisch sind, durch die Funktion `load_image_dataset_from_directory` verzerrt würden, damit sie in ein quadratisches Seitenverhältnis von 256x256 Pixel passen. Tests haben ergeben, dass die meisten Bilder des Datensatzes nur gerüfügig verzerrt werden. Einige Bilder besitzen jedoch signifikant mehr Pixel in der Höhe als in der Breite oder umgekehrt. Um dafür zu sorgen, dass alle Bilder ohne Verzerrung in das Modell gespeist werden, existiert der Parameter `crop_to_aspect_ratio`. Dieser Parameter sorgt dafür, dass das Bild derart zugeschnitten wird, dass es in das angegebene Bildformat passt. Hierbei wird stets der zentrale Teil des Bilds erhalten, während aus dem Rand des Bilds Teile abgeschnitten werden können. Da sich die Straßenschilder in den meisten Fällen mittig im Bild befinden, ist dies genau das gewünschte Verhalten.

Was die Funktion `load_image_dataset_from_directory` zurückgibt, ist ein `tf.data.Dataset` Objekt. Es kann somit direkt für die `CycleGAN.fit`-Methode verwendet werden. Ein letzter Schritt ist, die Bilder zu normalisieren. Dies geschieht mittels der Funktion `normalize_dataset` aus dem eigens definierten Modul `utils.load_data`. Diese Funktion normalisiert die Pixelwerte der Bilder auf den Bereich von -1 bis 1. Dies ist notwendig, um die Bilder in die KNNs einzuspeisen.

Damit ist das Laden der Trainingsdaten abgeschlossen. Die Piktogramme werden unter der Verwendung des gleichen Schemas geladen. Es ergibt sich ein `tf.data.Dataset` Objekt mit dem Namen `x_pictograms_processed`.

4.3.2 Ausführen des Trainings

Das Training wird vollständig durch die Klasse `CycleGan` durchgeführt. Für den Aufruf mit den gelandeten Datensatz-Objekten sind folgende Codezeilen notwendig:

```
cycle_gan = model.CycleGan(config)
cycle_gan.restore_latest_checkpoint_if_exists()
cycle_gan.fit(x_pictograms_processed, x_train_processed,
    epochs=config['training']['number_of_epochs'])
```

Listing 4.8: `train.py` - Laden des Trainingsdatensatzes

Das Skript `train.py` instanziert zunächst ein `CycleGan` Objekt. Falls vorherige Parameter in einem Checkpoint gespeichert sind, werden diese anschließend geladen. Ansonsten werden die Parameter durch TensorFlow zufällig initialisiert. Abschließend erfolgt der Aufruf der `fit`-Methode mit der in der Konfigurationsdatei angegebenen Anzahl an Epochen.

4.3.3 Logging

```
$ tensorboard --logdir ./logs/unet
$ tensorboard --logdir ./logs/resnet
```

4.3.4 Vorgehensweise

4.4 Trainingsergebnisse

Für das Training der U-Net- und ResNet-basierten CycleGANs dienen zwei verschiedene Systeme. Dabei zum einen Google Colab. Hier bietet Google eine kostenfreie Version sowie ein Premium-Abonnement an. In der kostenfreien Version ist jedoch kein Training über Nacht möglich, da das System nach einer etwa zwanzig-minütigen Inaktivität die Verbindung zum Rechner trennt. Unter anderem aus diesem Grund wird das Training zusätzlich auf einem Server der DHBW durchgeführt. Dieser besitzt eine Grafikkarte mit 25 Gigabyte Speicher und ist damit leistungsstärker als die von Google Colab angebotenen Systeme. Die Trainingsdauer pro Epoche ist für die trainierten Modelle in der nachfolgenden Tabelle aufgeführt. Es zeigt sich hier außerdem, dass das U-Net signifikant schneller trainiert als das ResNet, jedoch Checkpoints besitzt, die mehr Speicherplatz verbrauchen.

Trainingsdauer pro Epoche				
Modell	Google Colab	DHBW Server	Parameter	Checkpoint Größe
U-Net	30 min.	5 min.	0	1.340.240 Byte
ResNet	90 min.	30 min.	0	331.709 Byte

Tabelle 4.2: Vergleich von UNet und ResNet

4.4.1 U-Net

Das U-Net-basierte CycleGAN verbessert sich während des Trainings nahezu kontinuierlich. Das Modell ist mit einer Anzahl von 200 Epochen trainiert. Der Verlauf der Verlustfunktionen über die Anzahl der Trainingsschritte ist im Anhang in Abbildung A.2 gezeigt. Ein Trainingsschritt entspricht einem Durchlauf der `train_step` Funktion des CycleGAN über einem Batch. Es lassen sich verschiedene Dinge aus den Graphen ablesen: Die Verluste der Generatoren scheinen gegen einen Wert zu konvergieren. Die Verluste der Diskriminatoren zeigen hingegen deutliche Schwankungen ohne ein erkennbares Muster. Die Verluste der Generatoren sind beinahe um einen Faktor 10 größer als die der Diskriminatoren. Aus diesem Grund konvergiert der Gesamtverlust des CycleGAN gegen einen Wert. Dieser liegt bei 6.

Abbildung A.3 im Anhang zeigt zudem für die Epochen 1 bis 100, wie sich die Qualität der Generierung mit den Trainingsepochen verbessert. Hier lässt sich außerdem erkennen, dass das Modell durch die Wahl einer Bild-zu-Bild-Übersetzung die Schilder nicht selber erzeugen muss. Verschiedenen Trainingsdurchläufe haben ergeben, dass der Identity Loss für das UNet keinen signifikanten Einfluss hat. Damit kann argumentiert werden, dass er für dieses CycleGAN nicht nötig sei.

Die generierten Bilder zeigen verschiedene Hintergründe. Diese Varianz an Hintergründen ist allgemein pro Kategorie von Straßenschild gleich. Somit besitzen beispielsweise alle Schilder der Kategorie Geschwindigkeitsbegrenzung die gleichen Arten von Hintergründen. Die Schilder der Kategorie Aufhebung können allgemein als wenig fotorealistisch bewertet werden. Das ist vermutlich auf die geringere Anzahl an Trainingsdaten für diese Klassen zurückzuführen.

Wurde jetzt auf 200 Epochen trainiert. Hier scheint der total loss gegen einen Wert zu konvergieren. Die Qualität der Schilder sieht etwas besser aus als nach 150 Epochen. Die Hintergründe sind kaum besser. In der Evaluation den Unterschied der Prozentzahl von 150 Epochen zu 200 Epochen zeigen.

4.4.2 ResNet

Für das ResNet-basierte CycleGAN zeigen die Verlustfunktionen einen ähnlichen Verlauf wie bei dem U-Net-basierten CycleGAN. Aus diesem Grund sind hierfür die Graphen nicht im Anhang abgebildet. Der Unterschied ist, dass der Verlauf der Verlustfunktionen hier eine geringere Aussagekraft für die Qualität der generierten Bilder zu haben scheint. Das Training des ResNet-basierten Modells ist oszillierend, da das CycleGAN einige Klassen in nachfolgenden Epochen besser erzeugt, während andere Klassen eine schlechtere Generierungsqualität als davor aufweisen. Um diesem Verhalten entgegenzuwirken, verwendet dieses CycleGAN-Modell eine \mathcal{L}_2 Verlustfunktion für den Adversarial Loss, während das U-Net-basierte Modell weiterhin eine logarithmische Verlustfunktion verwendet. Auch diese Veränderung der Verlustfunktion, die laut der Literatur das Training stabilisieren kann, behebt das oszillierende Verhalten nicht.

Eine weitere Eigenschaft dieses Modells ist, dass bei 200 Epochen ein Modal Collaps auftritt. Das ist in Abbildung 4.1 dargestellt, wobei Epochen mit *Ep.* abgekürzt ist. Für jedes Piktogramm und jede Perspektive erzeugt das Modell einen beinahe identischen Hintergrund. Das gibt Hinweise darauf, dass die Generatoren die Diskriminatoren derart überlisten, dass letztere nicht mehr lernen.



Abbildung 4.1: Modal Collaps des ResNet nach 200 Trainingsepochen

Eine Lösung ist, das Training vorzeitig abzubrechen (*engl.: early stopping*). Diese Lösung wird gewählt. Dabei muss für jede infrage kommende Epochen manuell geprüft werden, welche davon das zufriedenstellendste Ergebnis zeigt. Auch da die Werte der Verlustfunktionen hierauf, wie bereits erwähnt, nur eine begrenzte Aussagekraft haben. Jede der Epochen 120 bis 190 erzeugt eine Bandbreite an Generierungsqualitäten. Abbildung ?? zeigt positiv herausstechende Bilder für verschiedene Epochen, während 4.3 negativ herausstechende Bilder zeigt. Die finale Entscheidung ist, das auf 170 Epochen trainierte ResNet-basierte Modell zu verwenden.



Abbildung 4.2: Positiv herausstechende Bilder des ResNets verschiedener Epochen

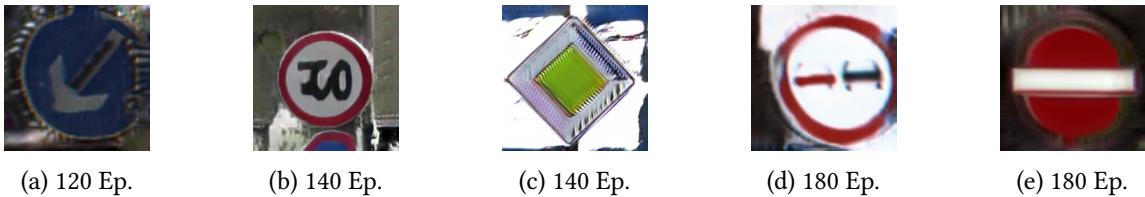


Abbildung 4.3: Negativ herausstechende Bilder des ResNets verschiedener Epochen

Eine zweite Lösung im Gegensatz zum vorzeitigen Abbruch des Trainings ist, dass die Generatoren ein anderes ResNet-Modell verwenden. Ein Modell, das verhindern kann, dass die Generatoren die Diskriminatoren vollständig überlisten. Aus diesem Grund besitzen die Generatoren in einem nächsten Trainingsdurchlauf 6 Residual Blocks, statt der durch die CycleGAN Veröffentlichung vorgeschlagene Anzahl an 9 für Bilder der Auflösung 256x256. Während hier kein Modal Collaps auftritt, ist das Training weiterhin oszillierend. Auch hier zeigt die Epoche 200 nicht die höchste Generierungsqualität. Bei diesem ResNet-basierten Modell wird deshalb Epoche 150 für die Evaluation in Kapitel 6 gewählt. Damit soll geprüft werden, welche Anzahl an Residual Blocks sich eignet. Die Abbildung ?? zeigt Beispielebilder für verschiedene Epochen.



Abbildung 4.4: Beispielbilder des ResNet-basierten Modells mit 6 Residual Blocks

4.5 Generierung

Das Skript `generate.py` dient dazu, Bilder von Straßenschildern mittels eines trainierten CycleGAN Modells zu generieren. Eine Design-Entscheidung ist hierbei, dass das Skript vollständig mittels der Kommandozeile konfiguriert werden kann. Dies folgt dem Leitprinzip dieser Studienarbeit, dass Anwender keinen Python-Code anpassen müssen. Das Skript besitzt folgende Kommandozeilenargumente:

Argument	Parameter	Aufgabe
--num-imgs	Ganzzahl	Anzahl der zu generierenden Bilder
--model	'unet' oder 'resnet'	Art des Modells
--motion-blur	-	Fügt Bewegungsunschärfe hinzu
--make-invalid	-	Markiert Schilder als ungültig
--snow	-	Fügt Schnee hinzu

Tabelle 4.3: Auswahl an Methoden aus der CycleGAN Klasse

Dabei können die Parameter `--motion-blur`, `--make-invalid` und `--snow` miteinander kombiniert werden. Beispielhafte Aufrufe des Skripts sind:

```
$ python generate.py --num-imgs 10 --motion-blur
$ python generate.py --num-imgs 10 --model 'resnet' --make-invalid
$ python generate.py --num-imgs 10 --snow --motion-blur
$ python generate.py --num-imgs 50 --model 'unet'
```

Ein spezieller Anwendungsfall dieses Modells könnte der folgende sein: Anwender möchten nur für bestimmte Arten von Straßenschildern Bilder generieren, statt für alle 43. Beispielsweise nur für Stopp-Schilder. Das ist insbesondere dann relevant, wenn das Modell dazu genutzt werden soll, einen bestehenden Datensatz auszugleichen. Wenn etwa bestimmte Klassen unterrepräsentiert sind.

Das Skript `generate.py` erlaubt explizit das folgende Vorgehen: Anwender können aus dem Ordner, in dem sich die Bilder der Piktogramme befinden, alle Arten von Straßenschildern löschen, die nicht generiert werden sollen. Befindet sich in dem Ordner beispielsweise nur ein Piktogramm für Stopp-Schilder, dann werden auch nur Bilder von Stopp-Schildern generiert. Dafür kann zum Beispiel ein zweiter Ordner für die Piktogramme angelegt werden, der dann in der TOML-Konfigurationsdatei unter dem Wert `'pictograms'` innerhalb der Kategorie `'paths'` angegeben wird.

5 | Augmentation der generierten Bilder

Die Augmentation der generierten Bilder wird durch das Python Modul `utils.image_augmentation` implementiert.

5.1 Ungültige Straßenschilder

In Kapitel 2.1 ist bereits beschrieben, dass als ungültig markierte Schilder offenbar eine Herausforderung für heutige Straßenschilderkennungen darstellen können. Das Ziel dieser Studienarbeit ist, solche Fälle simulieren zu können. Aus diesem Grund ist dieser Anwendungsfall implementiert.

Beispiele für durch dieses Projekt erzeugte, ungültige Straßenschilder sind in der nachfolgenden Abbildung dargestellt:

Bei der Umsetzung bieten sich zwei Möglichkeiten. Zum einen kann das CycleGAN darauf trainiert werden, solche Bilder eigenständig zu generieren. Dafür würden Trainingsdaten benötigt, die solche Schilder zeigen. Der Datensatz müsste somit um weitere Bilder ergänzt werden. Weitere reale Bilder hinzuzufügen ist nicht ohne weiteres möglich. Es wäre jedoch auch denkbar, bereits vorhandene Trainingsbilder mit einer Bildbearbeitungssoftware so anzupassen, dass die ungültige Schilder zeigen.

Eine weitere Möglichkeit ist, das Kreuz, das die Ungültigkeit eines Schildes markiert, nachträglich in die generierten Bilder einzufügen. Die Schwierigkeit ist hierbei, dass die Straßenschilder zufällig rotiert und skaliert sind. Das Kreuz muss so transformiert werden, dass es sich stets zentral und mit einer angepassten Rotation auf dem Schild befindet.

5.2 Bewegungsunschärfe

Verwackelte Bilder können insbesondere dann entstehen, wenn sich das Fahrzeug mit einer hohen Geschwindigkeit bewegt. Hierbei kann eine sogenannte Bewegungsunschärfe auftreten. Dies tritt bei einer Kamera auf, wenn sich das Bild während der Belichtungszeit deutlich verändert. Wenn also die Geschwindigkeit des Fahrzeugs groß ist im Vergleich zur Belichtungszeit.

Darauf eingehen, dass man das auch mit einer Fourier Transformation umgesetzen kann

Eine Bewegungsunschärfe kann mittels einer Faltung des Bildes mit einer Faltmatrix realisiert werden.

5.3 Schnee

Die Generierung von Schnee erfolgt in mehreren Schritten:

1. Erstelle ein Bild, das aus zufälligen schwarzen und weißen Pixeln besteht. Die Anzahl an weißen Pixel soll kleiner sein als die der schwarzen Pixel.
2. Führe auf dem Bild ein Gaußsches Weichzeichen aus. Hierdurch verschmieren die einzelnen weißen Pixel zu größeren Punkten.
3. Führe auf dem Bild eine Bewegungsunschärfe aus. Dadurch wird die Bewegung der Schneeflocken entlang einer Windrichtung simuliert.
4. Mache den schwarzen Hintergrund transparent und füge das erstellte Bild auf ein generiertes Straßenschild-Bild ein.

TODO: Datei einbinden und so linenumbers fixen

```
def add_snow(img_tensor, snow_intensity, motion_blur_intensity,  
            ↵ motion_blur_direction, p_snowflake_min=0.02,  
            ↵ p_snowflake_max=0.5):
```

Listing 5.1: Hinzufügen von Schnee: Funktionsdeklaration

Test Test

6 | Evaluation

Die Aufgabe der Evaluation ist prinzipiell folgende: Es soll gemessen werden, wie ähnlich die Wahrscheinlichkeitsverteilung der generierten Bilder zu der Verteilung der Trainingsbilder ist. Außerdem soll bestimmt werden, wie realistisch die generierten Bilder sind.

6.1 Evaluation der Generierung

Trainingsdatensatz	Trainingsbilder (#)	Testgenauigkeit (%)
Präparierter GTSRB	4.554	82
Gesamte Trainingsdaten	5.685	83
U-Net	4.300	62
ResNet (6 residual blocks)	4.300	46
ResNet (9 residual blocks)	4.300	53
Augmentierte Piktogramme	4.300	19
Piktogramme	43	12
Gemischt		

Tabelle 6.1: Ergebnisse des Trainings eines VGG16 Klassifikators

6.2 Evaluation der Augmentierung

6.3 Verbesserungsmöglichkeiten

- Bestimmte Funktionen in TensorFlow Graphen umwandeln; Dadurch Performance der Implementierung verbessern

7 | Zusammenfassung

Literatur

- [1] M. Staron, „AUTOSAR (AUTomotive Open System ARchitecture),“ in *Automotive Software Architectures: An Introduction*. Cham: Springer International Publishing, 2021, 97ff. ISBN: 978-3-030-65939-4. doi: [10.1007/978-3-030-65939-4_5](https://doi.org/10.1007/978-3-030-65939-4_5).
- [2] EU-Kommission, *Regulation (EU) 2019/2144 of the European Parliament and of the Council*, Art. 6 Abs. 2c, 5. Sep. 2021.
- [3] H. Ippen und M. Bach, „Verkehrsschild-Erkennung Test,“ *Autozeitung*, 2. Apr. 2019.
- [4] A. Gudigar, S. Chokkadi und R. U, „A review on automatic detection and recognition of traffic sign,“ *Multimedia Tools and Applications*, Jg. 75, S. 333–364, 2016. doi: [10.1007/s11042-014-2293-7](https://doi.org/10.1007/s11042-014-2293-7).
- [5] J. Stallkamp et al., „The German Traffic Sign Recognition Benchmark: A multi-class classification competition,“ in *IEEE International Joint Conference on Neural Networks*, 2011, S. 1453–1460.
- [6] H. Lengyel und Z. Szalay, „Test Scenario for Road Sign Recognition Systems with Special Attention on Traffic Sign Anomalies,“ in *2019 IEEE 19th International Symposium on Computational Intelligence and Informatics*, 2019, S. 000 193–000 198. doi: [10.1109/CINTI-MACRo49179.2019.9105238](https://doi.org/10.1109/CINTI-MACRo49179.2019.9105238).
- [7] I. Goodfellow, Y. Bengio und A. Courville, *Deep Learning*. MIT Press, 2016, <http://www.deeplearningbook.org>. (besucht am 11.05.2023).
- [8] D. Sonnet, *Neuronale Netze Kompakt, Vom Perceptron zum Deep Learning*. Wiesbaden: Springer Vieweg Wiesbaden, 2020. doi: [10.1007/978-3-658-29081-8](https://doi.org/10.1007/978-3-658-29081-8).
- [9] A. S. Glassner, „Deep Learning: A Visual Approach,“ in San Francisco: No Starch Press, 2021, ISBN: 978-1-7185-0072-3.
- [10] Y. Ho und S. Wookey, „The Real-World-Weight Cross-Entropy Loss Function: Modeling the Costs of Mislabeling,“ *IEEE Access*, Jg. 8, S. 4806–4813, 2020. doi: [10.1109/ACCESS.2019.2962617](https://doi.org/10.1109/ACCESS.2019.2962617).
- [11] K. O’Shea und R. Nash, *An Introduction to Convolutional Neural Networks*, 2015. doi: [10.48550/ARXIV.1511.08458](https://arxiv.org/abs/1511.08458).
- [12] M. T. Cicero, „Generatives Deep Learning,“ in übers. von M. Fraaß und K. Mach. Heidelberg: O'Reilly Verlag, 2020, Kap. Einführung ins Generative Deep Learning, ISBN: 9781492041948.

- [13] A. Karpathy, P. Abbeel, G. Brockman u. a., *Generative Models*, <https://openai.com/blog/generative-models/>, 16. Juni 2016. (besucht am 13.01.2023).
- [14] S. I. Nikolenko, „Generative Models in Deep Learning,“ in *Synthetic Data for Deep Learning*. Cham: Springer International Publishing, 2021, S. 97–137. doi: [10.1007/978-3-030-75178-4_4](https://doi.org/10.1007/978-3-030-75178-4_4).
- [15] A. van den Oord, N. Kalchbrenner und K. Kavukcuoglu, „Pixel Recurrent Neural Networks,“ *CoRR*, 2016. doi: [10.48550/arXiv.1601.06759](https://doi.org/10.48550/arXiv.1601.06759).
- [16] M. T. Jones, *Recurrent neural networks deep dive*, <https://developer.ibm.com/articles/cc-cognitive-recurrent-neural-networks>, 16. Aug. 2017. (besucht am 15.01.2023).
- [17] A. Oussidi und A. Elhassouny, „Deep generative models: Survey,“ in *2018 International Conference on Intelligent Systems and Computer Vision (ISCV)*, 2018, S. 1–8. doi: [10.1109/ISACV.2018.8354080](https://doi.org/10.1109/ISACV.2018.8354080).
- [18] D. Bank, N. Koenigstein und R. Giryes, „Autoencoders,“ *CoRR*, 2020. doi: [10.48550/arXiv.2003.05991](https://doi.org/10.48550/arXiv.2003.05991).
- [19] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza u. a., *Generative Adversarial Networks*, 2014. doi: [10.48550/ARXIV.1406.2661](https://doi.org/10.48550/ARXIV.1406.2661).
- [20] I. Goodfellow, J. Pouget-Abadie, M. Mirza u. a., „Generative Adversarial Networks,“ *Commun. ACM*, Jg. 63, Nr. 11, 139–144, 2020. doi: [10.1145/3422622](https://doi.org/10.1145/3422622).
- [21] J.-Y. Zhu, T. Park, P. Isola und A. A. Efros, *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*, 2017. doi: [10.48550/ARXIV.1703.10593](https://doi.org/10.48550/ARXIV.1703.10593).
- [22] TensorFlow, *CycleGAN*, <https://www.tensorflow.org/tutorials/generative/cyclegan>, o.D. (besucht am 01.04.2023).
- [23] Z. He, *CycleGAN-Tensorflow-2*, 13. Dez. 2021. Adresse: <https://github.com/LynnHo/CycleGAN-Tensorflow-2> (besucht am 06.05.2023).
- [24] K. He, X. Zhang, S. Ren und J. Sun, *Deep Residual Learning for Image Recognition*, 2015. arXiv: [1512.03385 \[cs.CV\]](https://arxiv.org/abs/1512.03385).
- [25] P. Isola, J.-Y. Zhu, T. Zhou und A. A. Efros, *Image-to-Image Translation with Conditional Adversarial Networks*, 2018. arXiv: [1611.07004 \[cs.CV\]](https://arxiv.org/abs/1611.07004).
- [26] O. Ronneberger, P. Fischer und T. Brox, *U-Net: Convolutional Networks for Biomedical Image Segmentation*, 2015. arXiv: [1505.04597 \[cs.CV\]](https://arxiv.org/abs/1505.04597).
- [27] D. Christine et al., „Synthetic Data generation using DCGAN for improved traffic sign recognition,“ *Neural Computing and Applications*, S. 1–16, Apr. 2021. doi: [10.1007/s00521-021-05982-z](https://doi.org/10.1007/s00521-021-05982-z).

- [28] D. Spata, D. Horn und S. Houben, „Generation of Natural Traffic Sign Images Using Domain Translation with Cycle-Consistent Generative Adversarial Networks,“ in *2019 IEEE Intelligent Vehicles Symposium (IV)*, 2019, S. 702–708. doi: [10.1109/IVS50000.2019.900000](https://doi.org/10.1109/IVS50000.2019.900000).
- [29] G. Nguyen, S. Dlugolinsky, M. Bobák u. a., „Machine Learning and Deep Learning Frameworks and Libraries for Large-Scale Data Mining: A Survey,“ *Artif. Intell. Rev.*, Jg. 52, Nr. 1, 77–124, 2019. doi: [10.1007/s10462-018-09679-z](https://doi.org/10.1007/s10462-018-09679-z).
- [30] TensorFlow, *tf.data.Dataset*, https://www.tensorflow.org/api_docs/python/tf/data/Dataset, o.D. (besucht am 30.03.2023).
- [31] P. Singh und A. Manure, *Learn TensorFlow 2.0, Implement Machine Learning and Deep Learning Models with Python*. Berkeley, CA: Apress, 2020. doi: [10.1007/978-1-4842-5558-2_1](https://doi.org/10.1007/978-1-4842-5558-2_1).
- [32] L. Huang, *Chinese Traffic Sign Database: Traffic Sign Recognition Database*, <http://www.nlpr.ia.ac.cn/pal/trafficdata/recognition.html>, o.D. (besucht am 25.03.2023).
- [33] United Nations Economic Commission for Europe, *Convention on Road Traffic*, 28. März 2006.
- [34] C. Ertler, J. Mislej, T. Ollmann, L. Porzi, G. Neuhold und Y. Kuang, *The Mapillary Traffic Sign Dataset for Detection and Classification on a Global Scale*, 2020. doi: [10.48550/arXiv.1909.04422](https://arxiv.org/abs/1909.04422).
- [35] R. Timofte, K. Zimmermann, und L. van Gool, „Multi-view traffic sign detection, recognition, and 3D localisation,“ *Journal of Machine Vision and Applications (MVA 2011)*, 2011. doi: [10.1007/s00138-011-0391-3](https://doi.org/10.1007/s00138-011-0391-3).
- [36] *Road Signs Dataset*, o.D. Adresse: <https://makeml.app/datasets/road-signs>.
- [37] J. Valentin und S. Bouaziz, *Introducing TensorFlow Graphics: Computer Graphics Meets Deep Learning*, https://www.tensorflow.org/api_docs/python/tf/keras/utils, o.D. (besucht am 31.03.2023).
- [38] OpenCV, *Introduction*, o.D. Adresse: <https://docs.opencv.org/4.x/d1/dfb/intro.html> (besucht am 08.05.2023).
- [39] J. A. Clark u.a., *Pillow, Overview*, o.D. Adresse: <https://pillow.readthedocs.io/en/stable/handbook/overview.html> (besucht am 08.05.2023).
- [40] S. Bond-Taylor, A. Leach, Y. Long und C. G. Willcocks, „Deep Generative Modelling: A Comparative Review of VAEs, GANs, Normalizing Flows, Energy-Based and Autoregressive Models,“ *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Jg. 44, Nr. 11, S. 7327–7347, 2022. doi: [10.1109/tpami.2021.3116668](https://doi.org/10.1109/tpami.2021.3116668).

- [41] Bundesanstalt für Straßenwesen (BASt), *Verkehrszeichen und Symbole, Verkehrszeichenkatalog 2017*, <https://www.bast.de/DE/Verkehrstechnik/Fachthemen/v1-verkehrszeichen/vz-start.html>, o.D. (besucht am 27.03.2023).
- [42] W. Burger und M. J. Burge, „Digital Image Processing: An Algorithmic introduction,“ in Springer Cham, 2022, Kap. Geometric Operations, S. 601–637. doi: [10.1007/978-3-031-05744-1](https://doi.org/10.1007/978-3-031-05744-1).
- [43] F. Dunn und I. Parberry, „3D Math Primer for Graphics and Game Development,“ in A K Peters/CRC Press, 2011, Kap. Rotation in Three Dimensions. Adresse: <https://gamedev.math.com/book/orient.html>.
- [44] S. Liu, „Study for Identity Losses in Image-to-Image Domain Translation with Cycle-Consistent Generative Adversarial Network,“ *Journal of Physics: Conference Series*, Jg. 2400, Nr. 1, S. 012030, 2022. doi: [10.1088/1742-6596/2400/1/012030](https://doi.org/10.1088/1742-6596/2400/1/012030).
- [45] R. Varma, *Downsides of the sigmoid activation and why you should center your inputs*, o.D. Adresse: <https://rohanvarma.me/inputnormalization> (besucht am 11.05.2023).
- [46] R. Varma, *TensorFlow Examples*, o.D. Adresse: <https://rohanvarma.me/inputnormalization> (besucht am 11.05.2023).
- [47] TensorFlow, *tf.map_fn*, o.D. Adresse: https://www.tensorflow.org/api_docs/python/tf/map_fn (besucht am 11.05.2023).
- [48] TensorFlow, *tf.keras.utils*, https://www.tensorflow.org/api_docs/python/tf/keras/utils, o.D. (besucht am 31.03.2023).

Anhang

A. Abbildungen

B. Listings

C. test

Abbildungen



Abbildung A.1: Alle 43 Piktogramme zu den generierten Klassen von Straßenschildern [41]

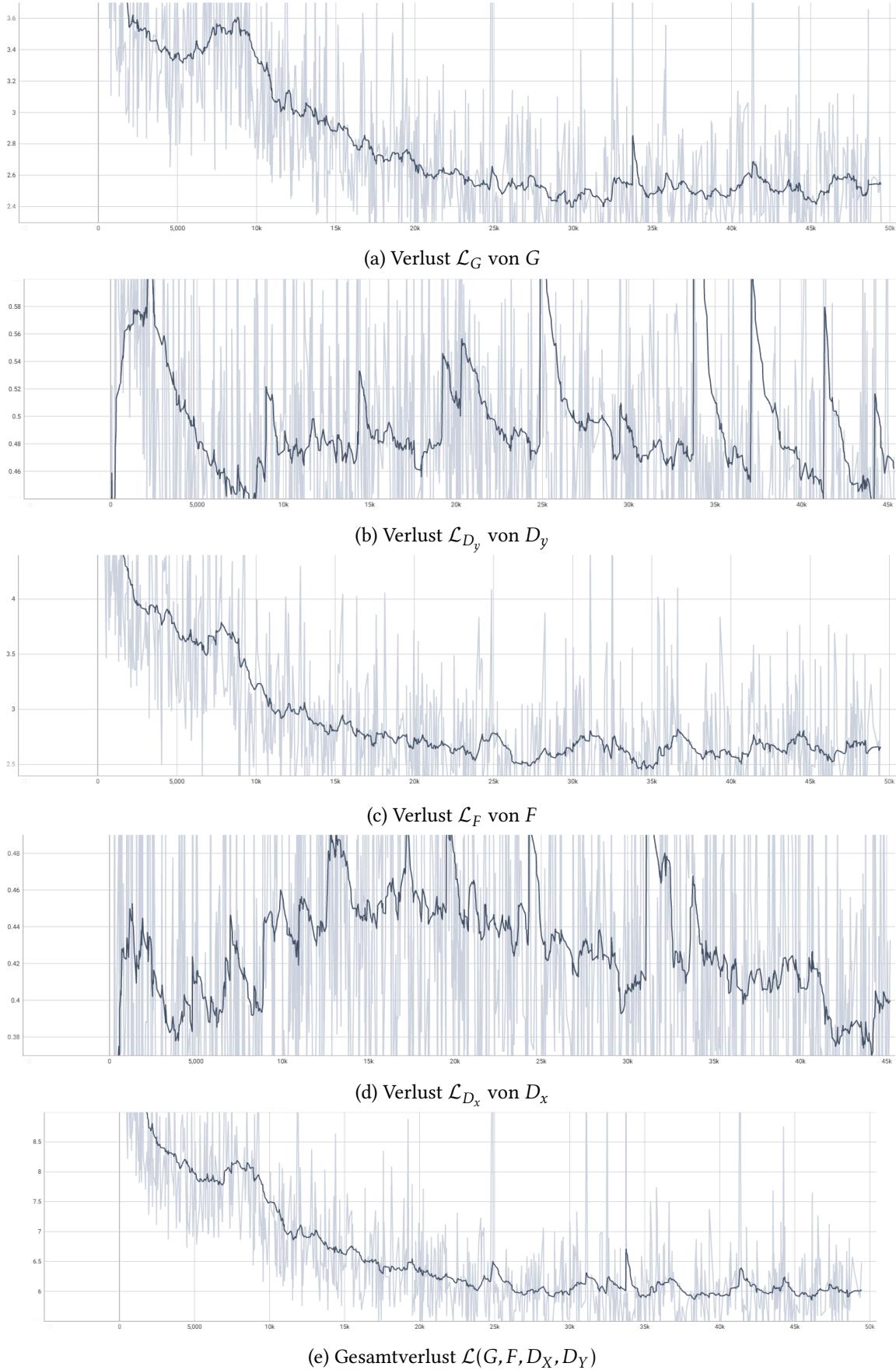


Abbildung A.2: Trainingsverlauf des U-Net bis Epoche 200 (x-Achse: Trainingsschritt, y-Achse: Verlust)



Abbildung A.3: Trainingsverlauf des U-Net-basierten CycleGAN bis Epoche 100

Listings

```
src
└── config
    └── config.toml

└── utils
    ├── __init__.py
    ├── image_augmentation.py
    ├── load_data.py
    ├── misc.py
    ├── preprocess_image.py
    ├── generate.py
    ├── model.py
    └── train.py
```

utils/preprocess_image.py

```
def randomly_transform_image_batch(img_tensor_batch,
                                   target_size=256):
    batch_size = img_tensor_batch.shape[0]

    # resize content
    min_content_size = target_size / 1.5
    # we have to work with default python lists because we need the
    # pop function
    content_sizes = [np.random.randint(low=min_content_size,
                                         high=target_size) for el in range(batch_size)]
    # copy of content_sizes; will be used to pop the elements
    content_sizes_tmp = content_sizes[:]
    transformed_imgs = tf.map_fn(lambda img:
                                 resize_content_of_img(img, target_size,
                                                       content_sizes_tmp.pop(0)), img)
```

```

# randomly rotate the image in x,y and z direction; scale values
  → are empirically chosen
bank_values = np.random.normal(loc=0.0, scale=3.5,
                                size=batch_size)
pitch_values = np.random.normal(loc=0.0, scale=0.01,
                                 size=batch_size)
heading_values = np.random.normal(loc=0.0, scale=0.01,
                                   size=batch_size)
transform_matrices = np.zeros((batch_size, 3, 3))
for i in range(batch_size):
    transform_matrices[i] =
        ↵ create_rotation_matrix(bank_values[i], pitch_values[i],
        ↵ heading_values[i])
transform_matrices = tf.convert_to_tensor(transform_matrices,
                                          dtype=tf.float32)

transformed_imgs = 1 - transformed_imgs
transformed_imgs =
    ↵ tfg_image_transformer.perspective_transform(transformed_imgs,
    ↵ transform_matrices)
transformed_imgs = 1 - transformed_imgs

return transformed_imgs, content_sizes, transform_matrices

```

Listing 1: Augmentierung eines Batches von Bildern

model.py

```

def fit(self, pictograms, real_images, epochs=1):
    """Train the model for a specific number of epochs. Checkpoints
    → are saved every epoch.

    Args:
        pictograms: 4d tensor containing the raw pictograms
        ↵ (batch_size, height, width, channels).
        real_images: 4d tensor containing the training images of
        ↵ street signs (batch_size, height, width, channels).
        epochs: Number of epochs to train the model.
    """
    print('Training...')
    with self.summary_writer.as_default():
        for epoch in range(epochs):
            self.total_epochs.assign_add(1) # increment

```

```

print(f'Epoch: {int(self.total_epochs)} /'
      f'{int(self.total_epochs) + epochs-(epoch+1)})')

# Single training step
for image_batch in tqdm(real_images):
    self.total_steps.assign_add(1) # increment

    # Transform the pictograms
    pictograms.shuffle(buffer_size=100,
                        reshuffle_each_iteration=True)
    single_pictogram_batch =
        pictograms.take(1).get_single_element()
    single_pictogram_batch, _, _ =
        utils.preprocess_image.randomly_transform_image_batch(
            single_pictogram_batch)

    # Train the model
    losses = self.train_step(single_pictogram_batch,
                            image_batch)

    # For Tensorboard; Log the losses
    for loss_name in losses:
        tf.summary.scalar(loss_name, losses[loss_name],
                           int(self.total_steps))

    # After each epoch: Generate an image
    pictograms.shuffle(buffer_size=100,
                        reshuffle_each_iteration=True)
    single_pictogram_batch =
        pictograms.take(1).get_single_element()
    single_pictogram_batch, _, _ =
        utils.preprocess_image.randomly_transform_image_batch(
            single_pictogram_batch)
    generator_test_result =
        self.generator_g(single_pictogram_batch)
    random_filename = str(uuid.uuid4())

    utils.misc.store_tensor_as_img(tensor=generator_test_result[0],
                                    filename=random_filename,
                                    relative_path='generated_images')

    self.summary_writer.flush() # write tensorboard logs to
                                log file

    if ((epoch + 1) % 10 == 0) and ((epoch + 1) < epochs):

```

```
    self.checkpoint_manager.save()
    print('Checkpoint saved for epoch
          ↪ {}'.format(epoch + 1))
self.checkpoint_manager.save()
print('Checkpoint saved for this training')
```

Listing 2: Augmentierung eines Batches von Bildern