

# **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](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](http://www.ib.dhbw-mannheim.de/fileadmin/ms/bwl-ib/Downloads_alt/Leitfaden_31.05.pdf), S. 49

# Inhaltsverzeichnis

<b>Abkürzungsverzeichnis</b>	<b>V</b>
<b>Abbildungsverzeichnis</b>	<b>VI</b>
<b>Tabellenverzeichnis</b>	<b>VII</b>
<b>Listings</b>	<b>VIII</b>
<b>1 Einleitung</b>	<b>1</b>
1.1 Problemstellung . . . . .	1
1.2 Vorgehensweise . . . . .	1
1.3 Ziel der Arbeit . . . . .	1
<b>2 Stand der Technik</b>	<b>2</b>
2.1 Straßenschilderkennung . . . . .	2
2.2 Künstliche Neuronale Netze . . . . .	4
2.2.1 Training . . . . .	6
2.2.2 Convolutional Neural Networks . . . . .	7
2.3 Bildgenerierung mittels Künstlicher Neuronaler Netze . . . . .	8
2.3.1 Mathematischer Hintergrund . . . . .	9
2.3.2 Pixel Recurrent Neural Networks . . . . .	10
2.3.3 Autoencoder . . . . .	13
2.3.4 Generative Adversarial Networks . . . . .	15
2.4 Vorherige Arbeiten . . . . .	18
2.4.1 Generierung Taiwanischer Straßenschilder mittels DCGAN . . . . .	18
2.4.2 Generierung Deutscher Straßenschilder mittels CycleGAN . . . . .	20
2.5 Machine Learning Frameworks . . . . .	21
2.5.1 TensorFlow . . . . .	21
2.5.2 Pytorch . . . . .	21
2.5.3 Weitere . . . . .	21
<b>3 Konzeption des Modells</b>	<b>22</b>
3.1 Datensatz . . . . .	22
3.2 Framework . . . . .	25
3.3 Architektur . . . . .	26
3.4 Datenaugmentation . . . . .	26
3.5 Training . . . . .	27
<b>4 Implementierung und Training</b>	<b>28</b>
4.1 Modelle . . . . .	28

4.2	Datenaugmentation . . . . .	29
4.2.1	Skalierung . . . . .	31
4.2.2	Rotation . . . . .	32
4.3	Training . . . . .	33
4.3.1	Laden der Datensätze . . . . .	33
4.3.2	Logging . . . . .	35
4.3.3	Ergebnisse . . . . .	36
4.4	Generierung . . . . .	36
<b>5</b>	<b>Augmentation der generierten Bilder</b>	<b>37</b>
5.1	Ungültige Straßenschilder . . . . .	37
5.2	Bewegungsunschärfe . . . . .	37
5.3	Schnee . . . . .	38
<b>6</b>	<b>Evaluation</b>	<b>39</b>
6.1	Evaluation der Generierung . . . . .	39
6.1.1	Farbhistogramme . . . . .	39
6.1.2	Wasserstein-Distanz . . . . .	39
6.1.3	Klassifizierung . . . . .	39
6.2	Evaluation der Augmentierung . . . . .	39
6.2.1	Mean Opinion Score . . . . .	39
6.3	Verbesserungsmöglichkeiten . . . . .	39
<b>7</b>	<b>Zusammenfassung</b>	<b>40</b>
<b>Anhang</b>		<b>44</b>

# Abkürzungsverzeichnis

<b>CNN</b>	Convolutional Neural Network
<b>CycleGAN</b>	Cycle-Consistent Generative Adversarial Network
<b>GAN</b>	Generative Adversarial Network
<b>GTSRB</b>	German Traffic Sign Recognition Benchmark
<b>KNN</b>	Künstliches Neuronales Netz
<b>PixelRNN</b>	Pixel Recurrent Neural Network
<b>SSIM</b>	Index struktureller Ähnlichkeit (eng.: Structural Similarity Index)
<b>SVM</b>	Support Vector Machine
<b>TOML</b>	Tomś Obvious Minimal Language

# Abbildungsverzeichnis

2.1	Erschwerende Einflüsse auf die Straßenschilderkennung . . . . .	3
2.2	Weitere mögliche Einflüsse auf die Straßenschilderkennung . . . . .	3
2.3	Einzelnes Neuron eines Künstliches Neuronales Netzs (KNNs) [7] . . . . .	5
2.4	Vollständiges KNN [7] . . . . .	5
2.5	Beispiel für eine Faltung (engl.: Convolution) [8] . . . . .	8
2.6	Beispielhafter Vergleich von $\hat{p}(x)$ und $p(x)$ [9] . . . . .	9
2.7	Taxonomie generativer Modelle [10] . . . . .	10
2.8	Bestimmung von $\hat{p}(x)$ mit PixelRNNs [11] . . . . .	11
2.9	Darstellung eines Recurrent Neural Networks [12] . . . . .	12
2.10	Architektur eines Autoencoders . . . . .	13
2.11	Zusammenspiel zwischen Generator und Discriminator . . . . .	15
2.12	Beispielergebnisse der Generierung taiwanischer SChilder [17] . . . . .	19
3.1	Beispielbilder aus dem GTSRB Datensatz [18] . . . . .	22
3.2	Häufigkeitsverteilung der Klassen von Straßenschildern im präparierten German Traffic Sign Recognition Benchmark (GTSRB) . . . . .	23
3.3	Beispielbilder aus der chinesischen Traffic Sign Recognition Database [20] . .	24
3.4	Beispielbild aus dem <i>Mapillary</i> Datensatz [22] . . . . .	24
3.5	Häufigkeitsverteilung der Kategorien von Straßenschildern im präparierten Datensatz . . . . .	25
3.6	Rotation der Straßenschilder mittels eulerscher Winkel . . . . .	27

# Tabellenverzeichnis

2.1	Vergleich der Objekterkennung mit und ohne künstliche Trainingsdaten . . . . .	19
2.2	Vergleich der Klassifikation mit echten und künstlichen Trainingsdaten [19] . .	21
4.1	Auswahl an Methoden aus der CycleGAN Klasse . . . . .	29
4.2	Vergleich von UNet und ResNet . . . . .	36

# Listings

4.1	Augmentierung eines Batches von Bildern . . . . .	31
4.2	Skalieren der Bild-Tensoren . . . . .	31
4.3	Augmentierung eines Batches von Bildern . . . . .	33
4.4	train.py - Laden des Trainingsdatensatzes . . . . .	34
5.1	Hinzufügen von Schnee: Funktionsdeklaration . . . . .	38

# **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

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]

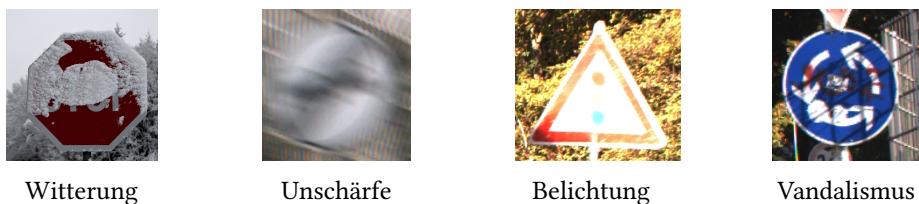


Abbildung 2.2: Weitere mögliche Einflüsse auf die Straßenschilderkennung

Ausgehend der Erkenntnisse der Tests aus dem genannten Artikel, kann in einigen solcher Situationen davon ausgegangen werden, dass sich in vielen Fahrzeugen nicht auf die automatische Erkennung der Schilder verlassen werden kann. Ein Ziel der Hersteller ist das Anbieten von Fahrzeugen, die vollständig autonom, das heißt ohne menschliches Eingreifen, fahren können. Damit dies möglich ist, muss die Software der Fahrzeuge auch solche Grenzfälle korrekt interpretieren. Dies 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. [5, 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 Jahrhunderts, 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. [6, 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. Letztere 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. [7]

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. [7]

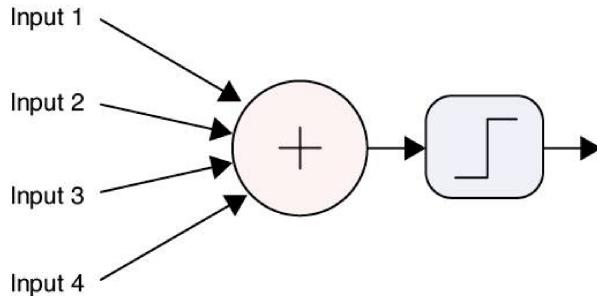


Abbildung 2.3: Einzelnes Neuron eines KNNs [7]

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. In der nachfolgenden Abbildung sind die Neuronen durch Kreise dargestellt und ihre Verbindungen durch Linien. 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. [6]

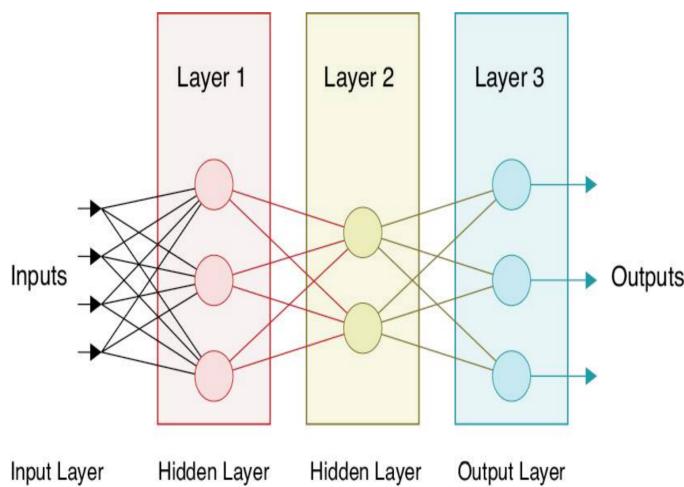


Abbildung 2.4: Vollständiges KNN [7]

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.

### 2.2.1 Training

Damit ein neuronales Netz trainiert werden kann, bedarf es einer Funktion, die die Qualität der Vorhersagen des Modells bestimmt. Erst dadurch kann verglichen werden, ob eine gegebene Änderung der Parameter eine Annäherung an das globale Optimum zur Folge hat. Diese Funktion wird als *Kostenfunktion* bezeichnet. Sie berechnet, gemittelt über alle  $m$  Trainingsbeispiele  $i$ , die Abweichung des vorhergesagten Wertes  $\hat{y}$  von dem tatsächlichen Wert  $y$ . Die Kostenfunktion in Abhängigkeit von der Gesamtheit der Gewichte und Biases  $\theta$  kann beispielsweise folgendermaßen definiert sein:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)})^2 \quad (2.1)$$

Anstelle des Terms  $\frac{1}{2}(\hat{y}^{(i)} - y^{(i)})^2$ , der die quadratische Abweichung der vorhergesagten von den erwarteten Werten beschreibt, können auch andere Metriken verwendet werden. Bezeichnet man dies verallgemeinert als Verlustfunktion  $L$  (*engl.: loss-function*), so kann die Kostenfunktion wie folgt definiert werden:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m L(\hat{y}^{(i)}, y^{(i)}) \quad (2.2)$$

Dies wird als überwachtes Lernen bezeichnet, da für die Berechnung der Kostenfunktion *gelabelte* Trainingsdaten erforderlich sind. Das bedeutet, dass jedes Trainingsbild die zu erwartende Ausgabe enthält. Bei Bildern für einen Katzenklassifikator muss beispielsweise für jedes Trainingsbild durch einen Menschen gekennzeichnet werden, ob es eine Katze enthält oder nicht. Erst so kann bewertet werden, inwieweit die jeweiligen Vorhersagen des Modells korrekt sind.

Es ist jedoch nicht nur erforderlich, die Qualität der Vorhersagen des Netzes zu bestimmen. Um die Vorhersagen in zukünftigen Iterationen zu verbessern, ist es notwendig, die Parameter des Modells zu verändern. Darunter fallen, wie bereits erwähnt, die Gewichte und Biases  $\theta$ . Von ihnen hängt der Wert der Kostenfunktion ab. Die Ausgangssituation ist hierbei folgende: Mit einem gegebenen  $\theta$  befindet man sich in einem bestimmten Punkt der Kostenfunktion  $J(\theta)$ . Gesucht ist eine Parameteränderung  $d\theta$ , mit der man sich 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.

Nähern tut man sich dem Optimum, indem man den Punkt  $\theta$  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 nähert man sich dem Optimum nur um einen kleinen Betrag. Diese Annäherung wird als Gradientenabstieg bezeichnet, während der Betrag der Annäherung durch eine sogenannte Lernrate  $\alpha$  bestimmt wird. 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. Es sind viele Trainingsschritte erforderlich, damit sich das Modell dem Optimum möglichst weit nähert. Der Gradientenabstieg muss demnach häufig durchgeführt werden.

Anhand der nachfolgenden Abbildung, soll der Gradientenabstieg visuell verdeutlicht werden.

### 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

Bildet 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:

$$1024 \cdot 1024 \cdot 3 = 3.145.728 \quad (2.3)$$

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 Faltungsmatrix. Jedes CNN besitzt dabei mitunter mindestens eine Schicht, die eine Faltung durchführt. Diese Schichten werden auch *Convolutional Layer* genannt. Ein Bestandteil eines jeden Convolutional Layers ist die Faltungsmatrix, welche auch *Kernel* genannt wird. Die Faltung besteht darin, dass diese Matrix über die Werte der Eingangsmatrix geschoben wird, und an jeder Position das Skalarprodukt der jeweiligen Pixelwerte mit den Parametern des Kernels berechnet wird. Dieses Ergebnis ergibt dann den jeweiligen Wert der Ausgangsmatrix.

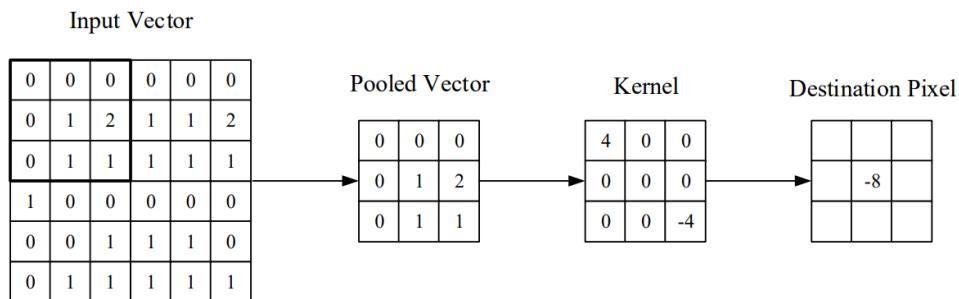


Abbildung 2.5: Beispiel für eine Faltung (engl.: Convolution) [8]

Zunächst wird der Kernel beispielsweise auf die Teilmatrix *links oben* der Eingangsmatrix angewendet. In Abbildung 2.5 ist dies visualisiert. 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 ein Label  $y$  auf einen gegebenen Eingang  $x$  zutrifft. Dafür benötigen sie annotierte Trainingsdaten.

Schachalgorithmen sollen hingegen lernen, Züge zu spielen, die die Gewinnchancen des Algorithmus maximieren. Eine weitere Art von KNNs sind *generative Netze*. Sie sollen lernen, neue Daten zu erzeugen, die der Verteilung der Trainingsdaten ähneln. Generative Netze zur Bildgenerierung sollen demnach anhand eines Trainingsdatensatzes lernen, welche Bilder sie erzeugen sollen. Dies lernen sie anhand der statistischen Verteilung der Trainingsdaten. Die Daten sind nicht annotiert, wodurch generative Netze in der Regel in das *Unüberwachte Lernen* einzuordnen sind.

### 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:

$$\hat{p}(x) \quad (2.4)$$

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.

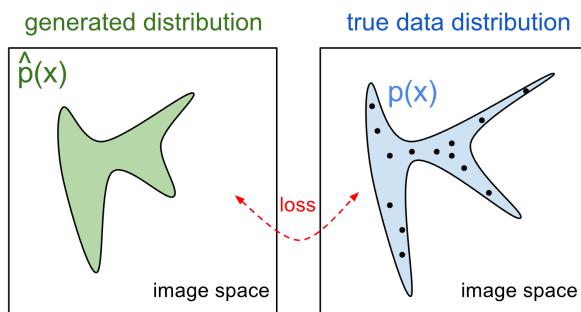


Abbildung 2.6: Beispielhafter Vergleich von  $\hat{p}(x)$  und  $p(x)$  [9]

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.

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.

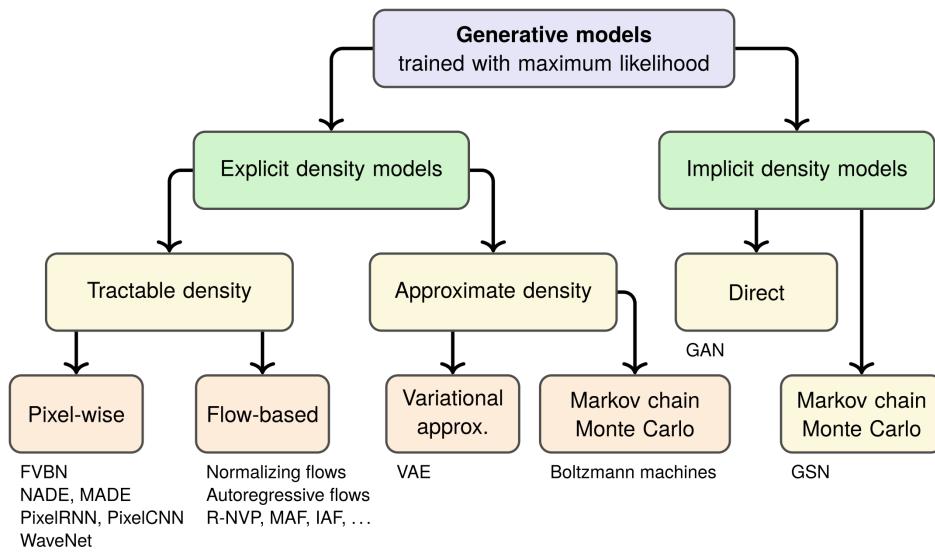


Abbildung 2.7: Taxonomie generativer Modelle [10]

Es soll in diesem Kapitel auf verschiedene Architekturen generativer Netze eingegangen werden. Generative Adversarial Networks (GANs) werden gesondert in Kapitel 2.3.3 beschrieben.

### 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. [11]

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 2.5 ist dabei dargestellt, wie die Wahrscheinlichkeit eines jeden Pixels in die gesamte Verteilung  $\hat{p}(x)$  einfließt. [11]

$$\hat{p}(x) = \hat{p}(x_1, x_2, \dots, x_{n^2}) = \prod_{i=1}^{n^2} \hat{p}(x_i | x_1, \dots, x_{i-1}) \quad (2.5)$$

Jeder Pixel  $x_i$  besitzt eine eigene Wahrscheinlichkeitsverteilung  $\hat{p}(x_i | x_1, \dots, x_{i-1})$ . Sie ist abhängig 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. [11]

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.

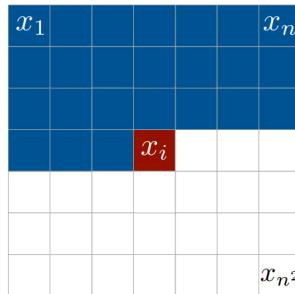


Abbildung 2.8: Bestimmung von  $\hat{p}(x)$  mit PixelRNNs [11]

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.

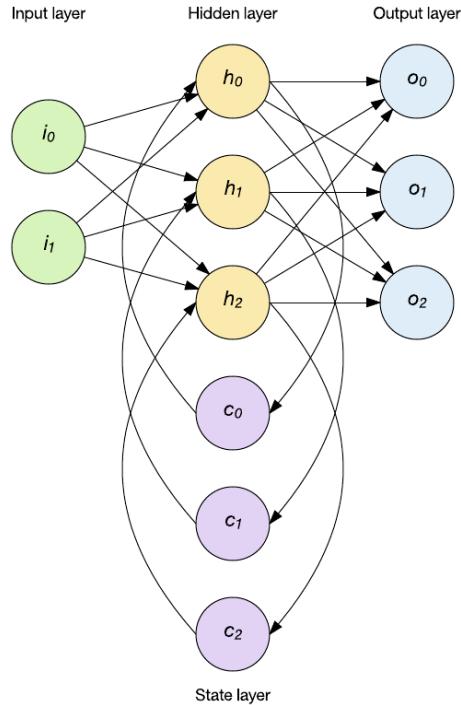


Abbildung 2.9: Darstellung eines Recurrent Neural Networks [12]

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. [13]

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. [13]

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. [11]

### 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.

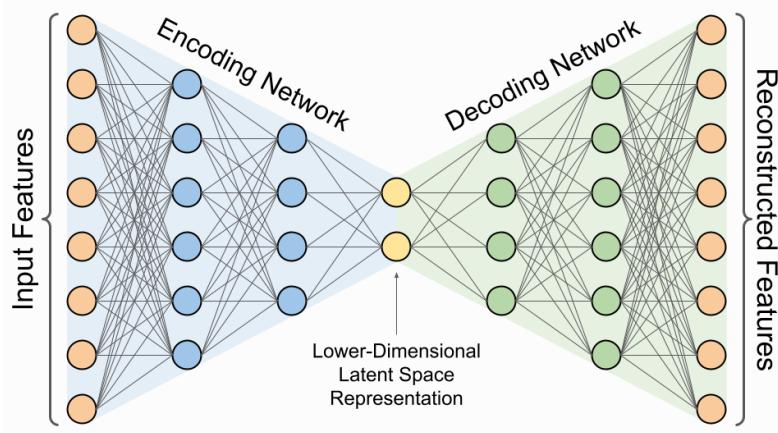


Abbildung 2.10: Architektur eines Autoencoders

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 latenden 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. [7]

**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:

$$p(z|x) \quad (2.6)$$

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  $\vec{z} = [z_1, z_2, \dots, z_n]^\top$  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?

[14]

### 2.3.4 Generative Adversarial Networks

Eine weitere Architektur, die zur Bildgenerierung verwendet werden kann, sind die sogenannten *Generative Adversarial Networks* (GANs). Sie wurden unter anderem von Ian Goodfellow im Jahre 2014 entwickelt.

Ein GAN besteht aus zwei Komponenten. Dem *Generator* und dem *Discriminator*. Der Generator erzeugt aus einem zufälligen Eingangsvektor ein Bild. Der Discriminator erhält ein Bild als Eingang und gibt entweder den Wert 0 oder den Wert 1 aus. Er soll erkennen, ob das gegebene Bild aus den Trainingsdaten stammt, oder ob es künstlich generiert wurde. Bei sowohl dem Generator als auch dem Discriminator handelt es sich um KNNs. Diese beiden Komponenten werden so miteinander gekoppelt, dass der Discriminator stets entweder ein erzeugtes Bild des Generators oder ein Bild aus den Trainingsdaten erhält. Dies ist in der nachfolgenden Abbildung beispielhaft dargestellt:

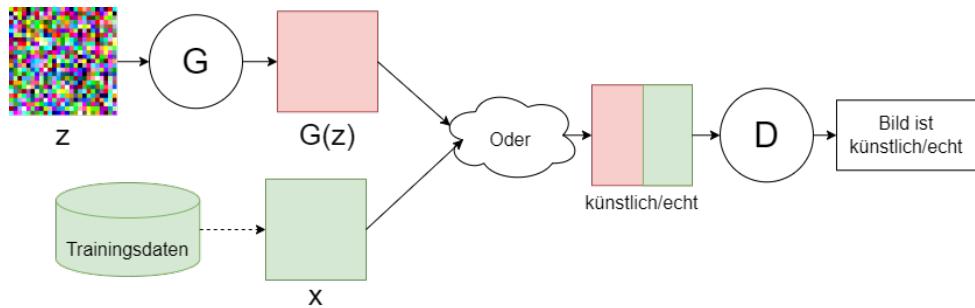


Abbildung 2.11: Zusammenspiel zwischen Generator und Discriminator

Der Generator  $G$  erhält einen zufälligen Eingangsvektor  $z$ . Letzterer kann als ein weißes Rauschen beschrieben werden. Das generierte Bild ist das Resultat der Funktion  $G(z)$  des Generators  $G$ . Anschließend wird dem Discriminator  $D$  entweder das generierte Bild  $G(z)$  oder ein Bild  $x$  aus den Trainingsdaten gezeigt. Der Ausgabewert des Discriminators ist daraufhin die Einschätzung, ob das gezeigte Bild echt oder künstlich generiert ist. Ziel des Generators ist, dass die Verteilung der künstlichen Daten  $p(G(z))$  möglichst ähnlich der Verteilung  $p(x)$  ist. Der Discriminator soll die beiden Verteilungen möglichst gut voneinander unterscheiden können. Somit handelt es sich um ein direktes Gegenspiel zwischen Generator und Discriminator. Sie versuchen sich gegenseitig zu überlisten.

**Training** Dies spiegelt sich auch im Training von GANs wider.  $V(D, G)$  stellt die zu optimierende Funktion dar: [15]

$$\min_G \max_D V(D, G) = \mathbb{E}_x[\log(D(x))] + \mathbb{E}_z[\log(1 - D(G(z)))] \quad (2.7)$$

Bei dem Training handelt es sich um ein Min-Max-Problem. Der Generator versucht die Funktion  $V(G, D)$  zu minimieren, wohingegen der Discriminator sie zu maximieren versucht. Der Term 2.8 stellt die Fehlerrate des Discriminators auf den echten Trainingsdaten dar. Mit anderen Worten: Wie viele echte Bilder er als unecht klassifiziert. Der Term 2.9 beschreibt, wie viele unechte Daten der Discriminator als echt klassifiziert. [7]

$$\mathbb{E}_x[\log(D(x))] \quad (2.8)$$

$$\mathbb{E}_z[\log(1 - D(G(z)))] \quad (2.9)$$

Der Term 2.8 ist nicht vom Generator abhängig, sondern lediglich von  $D(x)$ . Hierauf hat der Generator keinen Einfluss, wodurch er alleine durch diesen Teil der Kostenfunktion nicht trainiert wird. Der Discriminator besitzt somit zwei Terme, die ihn trainieren, während der Generator nur einen besitzt. Aus diesem Grund werden dem Discriminator 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 Discriminator so gut wie möglich identifizieren kann, ob ein gegebenes Bild aus  $p(x)$  stammt, während der Generator dennoch in der Lage ist, den Discriminator 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. [7]

Im praktischen Einsatz wird nur der Generator des GANs verwendet. Der Discriminator 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. [7]

Die bisher beschriebene Architektur von GANs wird auch als *Vanilla GAN* bezeichnet. Dies entspricht dem, wie GANs in Goodfellow's Publikation definiert werden [15]. 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 Discriminator überlisten kann und generiert es deshalb jedes Mal, egal welchen Input man ihm zuführt. Dies ist für diesen Anwendungsfall insofern von Relevanz, als dass der Generator verschiedene Arten von Straßenschildern generieren soll, wobei jedes Bild zusätzlich einen unterschiedlichen Hintergrund besitzen soll. Lösen lässt sich das Problem des *Modal Collaps* beispielsweise mit sogenannten Cycle-Consistent Generative Adversarial Networks (CycleGANs). [7]

**CycleGANs** Ein CycleGAN besteht aus zwei miteinander gekoppelten GANs. Diese werden klassischerweise als  $G$  und  $F$  bezeichnet. Bei  $G$  handelt es sich um ein Vanilla GAN, so wie in diesem Kapitel bisher beschrieben. Erweitert wird das Netzwerk jedoch so, dass es den Output von  $G$  an  $F$  weitergibt.  $F$  erhält somit das von  $G$  künstlich generierte Bild als Input. Aufgabe von  $F$  ist, daraus den ursprünglichen Input, der  $G$  zugeführt wurde, zu reproduzieren. Somit soll das gesamte Netzwerk nicht nur neue Bilder generieren können, sondern soll auch von einem generierten Bild zurück auf den zugeführten Input schließen können. Dies lässt sich mathematisch so darstellen, dass  $G$  und  $F$  folgende Abbildungen implementieren:

$$G : X \mapsto Y \wedge F : Y \mapsto \tilde{X} \quad (2.10)$$

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 Input  $X$  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.

CycleGANs sind für spezielle Anwendungsgebiete gedacht, in denen ausgewählte, und somit nicht-zufällige Eingabewerte verwendet werden. Dazu zählen Gebiete wie *Style Transfer* oder die Transformation von Bildern, respektive Bildelementen. Diese Studienarbeit stellt eine solche Problemstellung dar, da das auf dem generierten Bild gezeigte Straßenschild durch den Eingang des Netzwerks definiert wird.

Das Training von CycleGANs basiert auf mehreren Kostenfunktionen. Die GANs  $G$  und  $F$  besitzen jeweils einen eigenen *Adversarial Loss*. Damit ist die in Gleichung 2.7 beschriebene Kostenfunktion eines Vanilla GANs gemeint. Im Kontext von CycleGANs ist sie, für das GAN  $G$  wie folgt definiert:

$$L_{GAN}(G, D_Y, X, Y) = \mathbb{E}_y[\log D_Y(y)] + \mathbb{E}_x[\log(1 - D_Y(G(x)))] \quad (2.11)$$

Die Funktion für GAN  $F$  ist identisch, mit dem Unterschied, dass sie von  $F$  und  $D_X$  statt von  $G$  und  $D_Y$  abhängt.

Als weitere Kostenfunktion besitzen CycleGANs einen *Cyclic Loss* (Gleichung 2.12). 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.

$$L_{cyc}(G, F) = \mathbb{E}_x[\|F(G(x)) - x\|_1] + \mathbb{E}_y[\|G(F(y)) - y\|_1] \quad (2.12)$$

Um die gesamten Kosten des CycleGANs zu erhalten, werden die bisher beschriebenen Kostenfunktionen addiert. Der *Cyclic Loss*  $L_{cyc}(G, F)$  wird dabei mit einem absoluten Wert  $\lambda$  multipliziert, um die Kosten dieser Funktion im Vergleich zu den *Adversarial Losses* gewichten zu können. In der Veröffentlichung der CycleGANs wird ein  $\lambda$  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  $\lambda$  stellt einen Hyperparameter dar. [16]

$$L(G, F, D_X, D_Y) = L_{GAN}(G, D_Y, X, Y) + L_{GAN}(F, D_X, Y, X) + \lambda \cdot L_{cyc}(G, F) \quad (2.13)$$

[16]

## Wasserstein GANs 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. [17]

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. [17]

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: Beispieldaten der Generierung taiwanischer Schilder [17]

Zur Beurteilung der Generierung wird mitunter der sogenannte Index struktureller Ähnlichkeit (eng.: 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. [17]

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 beinhalten. Die Ergebnisse können folgender Tabelle entnommen werden: [17]

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. [17]

#### 2.4.2 Generierung Deutscher Straßenschilder mittels CycleGAN

Eine weitere Publikation, die sich mit der künstlichen Generierung von Bildern mit Straßen-schildern konzentriert, verwendet einen Datensatz, der deutsche Straßenschilder enthält. Er ist unter dem Namen GTSRB bekannt. Auf den Datensatz wird näher in Kapitel 3 eingeganen, 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. [18] [19]

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

[19]

#### 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 [7]. 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 ?? 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. [19]

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 [19]

## 2.5 Machine Learning Frameworks

### 2.5.1 TensorFlow

### 2.5.2 Pytorch

### 2.5.3 Weitere

# 3 | Konzeption des Modells

## 3.1 Datensatz

Analog zu der in Kapitel 2.4.2 beschriebenen Arbeit wird der GTSRB als Datensatz für diese Studienarbeit verwendet. 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 .1 dargestellt. Beispielbilder aus dem GTSRB befinden sich in der folgenden Abbildung: [18]



Abbildung 3.1: Beispielbilder aus dem GTSRB Datensatz [18]

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 verwendet werden. 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. [18]

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 wird 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. Dadurch wird 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.

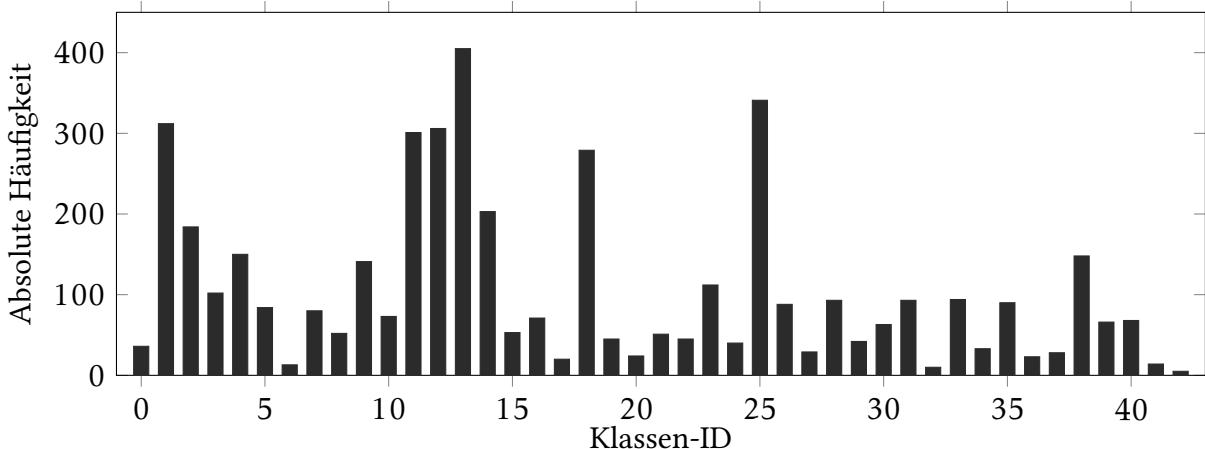


Abbildung 3.2: Häufigkeitsverteilung der Klassen von Straßenschildern im präparierten GTSRB

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 [19]. Aus diesem Grund, und da die Größe des präparierten Datensatzes als zu gering bewertet werden kann, wird 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. In dieser Studienarbeit wird dieselbe Kategorisierung verwendet, jedoch mit deutschen Klassennamen. 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 kann ein größerer Anteil des Datensatzes genutzt werden. Allgemein ähneln diese Bilder 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: [20]



Abbildung 3.3: Beispielbilder aus der chinesischen Traffic Sign Recognition Database [20]

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 [21]. [20]

Weiterhin 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 zu sehen sind. Die Datensätze nennen sich *Mapillary Traffic Sign Dataset* und *BelgianTS Dataset* [22] [23].



Abbildung 3.4: Beispielbild aus dem *Mapillary* Datensatz [22]

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 geringen Anteil aus. Aus einem dritten Datensatz werden weniger als 50 Bilder verwendet [24]. 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 Legende zu *Sonstige* zugeordnet.

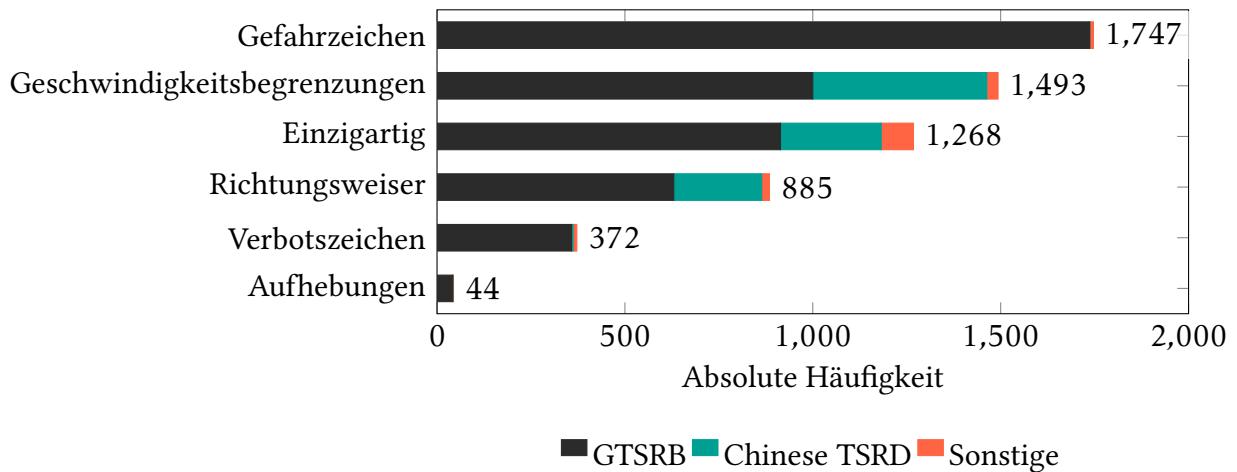


Abbildung 3.5: Häufigkeitsverteilung der Kategorien von Straßenschildern im präparierten Datensatz

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 verfolgt das Ziel, Straßenansichten für alle Straßen auf der Welt bereitzustellen. Hier können Benutzer eigene Bilder hochladen und den entsprechenden GPS-Koordinaten zuordnen. Einer der genannten Datensätze stammt aus einer Mischung von Mapillary-Bildern.

## 3.2 Framework

Seit Version 2.0 unterstützt TensorFlow die *EagerExecution*. Diese wird für die Implementierung der Studienarbeit genutzt. [25]

...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....

### Tensorflow Addons

**Tensorflow Graphics** Erklären: Wieso benutze ich, wo möglich, Tensorflow Graphics statt OpenCV? Wieso gibt es Tensorflow Graphics überhaupt?

**OpenCV** Erwähnen: OpenCV wird nur da benutzt, wo keine Tensorflow funktionen verwendet werden können. Beeinträchtigt die Performance.

### 3.3 Architektur

In Kapitel 2.3 werden verschiedene generative Netzwerkarchitekturen vorgestellt. Zunächst soll darauf eingegangen werden, welche dieser Herangehensweisen gewählt wird. Jede der Architekturen besitzt ihre Vor- und Nachteile. Der größte Nachteil von PixelRNNs ist, dass die Pixel eines Bildes nicht parallel zueinander generiert werden können. Dies verlangsamt die Generierung und damit das Training des Modells.

**Entscheidungsmatrix?**

[26]

### 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. [27] [28]

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

der nachfolgenden Grafik abgebildet. 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. [28]



Abbildung 3.6: 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. [28]

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

Überlicherweise soll beim Training von neuronalen Netzen die Verlustfunktion gegen den Wert *null* streben. Bei diesem Modell soll die Verlustfunktion jedoch gegen einen Wert konvergieren, der größer ist als *null*. In diesem Fall schaffen es weder die Generatoren, die Funktionswerte zu minimieren, noch können die Diskriminatoren die Funktionswerte weiter maximieren. In diesem Fall ist das sogenannte *Nash-Gleichgewicht* erreicht. Das Training ist somit beendet, da sich das Modell mit den gegebenen Daten nicht weiter verbessern kann.

# 4 | Implementierung und Training

Die Ordnerstruktur der Implementierung ist im Anhang abgebildet. Im wesentlichen wird in diesem Kapitel auf folgende Bestandteile eingegangen:

- 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

Es finden sich hierbei verschiedene Programmierparadigmen: objektorientierte Programmierung, prozedurale Programmierung und funktionale Programmierung. Das CycleGAN ist als Klasse implementiert und lässt sich somit innerhalb der Skripte instanziieren. Das Modul `utils`

Rotation der Piktogramme erklären

## 4.1 Modelle

Das Modell ist in der Datei `model.py` implementiert. Innerhalb dieser Datei existiert eine Klasse CycleGAN, die im wesentlichen folgende Methoden besitzt:

Methode	Aufgabe
<code>__init__</code>	Initialisieren der Attribute mittels der Konfigurationsdatei
<code>compile</code>	Kompilieren der Generator- und Diskriminatormodelle
<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_...<sup>1</sup></code>	Laden der aktuell gespeicherten Parameter des Modells

Tabelle 4.1: Auswahl an Methoden aus der CycleGAN Klasse

Das Modell objektorientiert zu implementieren scheint entgegen üblicher Konventionen zu stehen.

Um schnellere Ergebnisse zu erzielen, damit sich vor allem auf die Optimierung der Implementierung und der Augmentierung der generierten Bilder konzentriert werden kann, basiert der Inhalt dieser Klasse zum Teil auf einem Beispiel von TensorFlow [29]. Dieses Beispiel beinhaltet eine Implementierung eines CycleGANs. Der Quelltext dieses Beispiels wird für die Umsetzung des CycleGANs dieser Studienarbeit als Basis genommen und verändert, beziehungsweise erweitert. Es sei hierbei betont, dass jede Codezeile, die identisch mit dem Code aus dem Beispiel ist, vorher hinterfragt und bezüglich ihrer Sinnhaftigkeit für diesen Anwendungsfall getestet worden ist. Darauf soll explizit in diesem Kapitel eingegangen werden.

Wie bereits erwähnt, besteht ein CycleGAN aus vier KNNs: Generator X, Generator Y, Diskriminator X, Diskriminator Y.

## 4.2 Datenaugmentation

In Kapitel 3.4 ist das Vorgehen für die Datenaugmentation beschrieben. Hier soll auf die konkrete Implementierung dessen eingegangen werden.

Zur Augmentierung müssen die Piktogramme der Straßenschilder zufällig rotiert und skaliert werden. Hierzu soll ein Python-Framework genutzt werden. Prominente Beispiele hierfür sind *Pillow*, *OpenCV* und *Scikit-Image*. Besonders während des Trainings des Modells ist das Ziel dieser Studienarbeit, möglichst nur TensorFlow-eigene Funktionen zu verwenden. Das

---

<sup>1</sup> Vollständiger Methodename: `restore_latest_checkpoint_if_exists`

hängt damit zusammen, dass sie, hinsichtlich der Performance, explizit auf die Verwendung innerhalb von TensorFlow optimiert sind. Aus diesem Grund wird für die Augmentierung der Straßenschild-Piktogramme ausschließlich die Bibliothek *TensorFlow Graphics* eingesetzt. Sie erlaubt nicht nur das Transformieren einzelner Bilder, sondern eines gesamten Batches von Bildern. Frameworks, die nicht auf das maschinelle Lernen ausgelegt sind, erlauben dies nicht. Damit kann die Augmentierung mehrere Bilder parallel ausgeführt werden. [30]

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`. In dem nachfolgenden Listing ist diese Funktion dargestellt:

```

1  def randomly_transform_image_batch(img_tensor_batch,
2      ↵ target_size=256):
3          batch_size = img_tensor_batch.shape[0]
4
5          # resize content
6          min_content_size = target_size / 1.5
7          # we have to work with default python lists because we need the
8          ↵ pop function
9          content_sizes = [np.random.randint(low=min_content_size,
10              ↵ high=target_size) for el in range(batch_size)]
11          # copy of content_sizes; will be used to pop the elements
12          content_sizes_tmp = content_sizes[:]
13          transformed_imgs = tf.map_fn(lambda img:
14              ↵ resize_content_of_img(img, target_size,
15              ↵ content_sizes_tmp.pop(0)), img_tensor_batch)
16
17          # randomly rotate the image in x,y and z direction; scale values
18          ↵ are empirically chosen
19          bank_values = np.random.normal(loc=0.0, scale=3.5,
20              ↵ size=batch_size)
21          pitch_values = np.random.normal(loc=0.0, scale=0.01,
22              ↵ size=batch_size)
23          heading_values = np.random.normal(loc=0.0, scale=0.01,
24              ↵ size=batch_size)
25          transform_matrices = np.zeros((batch_size, 3, 3))
26          for i in range(batch_size):
27              transform_matrices[i] =
28                  ↵ create_rotation_matrix(bank_values[i], pitch_values[i],
29                  ↵ heading_values[i])
30          transform_matrices = tf.convert_to_tensor(transform_matrices,
31              ↵ dtype=tf.float32)
32
33          transformed_imgs = 1 - transformed_imgs
34          transformed_imgs =
35              ↵ tfg_image_transformer.perspective_transform(transformed_imgs,
36              ↵ transform_matrices)
```

```
23     transformed_imgs = 1 - transformed_imgs  
24  
25     return transformed_imgs, content_sizes, transform_matrices
```

Listing 4.1: Augmentierung eines Batches von Bildern

Die Funktion erhält einen vierdimensionalen Tensor `img_tensor_batch` als Eingang. Dieser Tensor beinhaltet den Batch an Bildern, die transformiert werden sollen. 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 generiert zunächst mittels `Numpy` eine Liste an zufälligen `content_sizes`. Anschließend folgt ein Codeabschnitt, der einer näheren Erläuterung bedarf:

```
1 # copy of content_sizes; will be used to pop the elements  
2 content_sizes_tmp = content_sizes[:]  
3 transformed_imgs = tf.map_fn(lambda img: resize_content_of_img(img,  
        ↴ target_size, content_sizes_tmp.pop(0)), img_tensor_batch)
```

Listing 4.2: Skalieren der Bild-Tensoren

Zunächst wird die Liste `content_sizes` in der Variable `content_sizes_tmp` dupliziert. Anschließend erfolgt die Skalierung der Bilder. Dabei wird die von TensorFlow bereitgestellte Funktion `tf.map_fn()` verwendet. Folgende Stichpunkte sollen ihre Funktionsweise beschreiben:

- **Parameter:** Eine Funktion und ein Tensor der Dimension  $n$
- **Zweck:** Führt die als Parameter übergebene Funktion auf jedem sub-Tensor der Dimension  $n - 1$  des übergebenen Tensor aus. Die Ausführung ist dabei parallel möglich. Anschließend werden die Tensoren wieder zu einem einzelnen Tensor der Dimension  $n$  zusammengefügt.

- **Rückgabe:** Ein Tensor der Dimension  $n$ , der alle zusammengefügten, gegebenenfalls veränderten sub-Tensoren enthält

In diesem Fall übergeben wir einen vierdimensionalen Tensor an die Funktion `tf.map_fn()`. Die Funktion macht daraus eine Menge an dreidimensionalen Tensoren, skaliert den Inhalt dieser Bilder und fügt sie wieder zu einem vierdimensionalen Tensor zusammen. 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 das hier als nicht erachtet wird, soll im folgenden geklärt werden.

Die Funktion, die für jedes Bild ausgeführt wird, nennt sich `resize_content_of_img`. Das ist keine TensorFlow-Funktion, sondern eine eigens definierte. Bei Bildern mit einem weißen, schwarzen oder transparenten Hintergrund dient Sie dazu, den Inhalt des Bilds zu skalieren, während das Bildformat identisch bleibt. Die Bilder der Piktogramme behalten ihre Größe von 256x256 Pixel bei, während das gezeigte Piktogramm entweder vergrößert oder verkleinert wird. Intern nutzt zwei TensorFlow Funktionen. Obwohl es möglich wäre, den vierdimensionalen Tensor an die TensorFlow Funktionen zu übergeben, erhält `resize_content_of_img` lediglich dreidimensionale Tensoren, sprich einzelne Bilder, als Parameter. Damit muss die Funktion für jedes Bild einzeln ausgeführt werden. Das hängt damit zusammen, dass die Piktogramme in dem Batch unterschiedliche Skalierungen besitzen sollen. Die TensorFlow Funktionen sind jedoch nur dazu in der Lage, eine bestimmte Skalierung auf alle Bilder des Batches auszuführen.

Ein Aspekt, auf den noch eingeganen werden soll, ist die Rolle der Liste `content_sizes_tmp`.

## 4.2.2 Rotation

Rotationen mittels eulerscher Winkel können mit Rotationsmatrizen durchgeführt werden. 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 [28]. Um daraus eine einzelne Rotationsmatrix  $R$  zu erhalten, werden sie miteinander multipliziert. Dazu dient die Funktion `create_rotation_matrix`.

Vor der Erstellung der Rotationsmatrizen  $R$ , muss die Funktion `randomly_transform_image_batch` zufällige Winkel  $(\alpha_x, \alpha_y, \alpha_z)$  erzeugen. Zur Erzeugung der Zufallszahlen wird auch hier Numpy genutzt. 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.

Nachfolgend ist die zufällige Generierung der Rotationswinkel  $\alpha_z$  gezeigt:

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

Listing 4.3: 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 nicht ein Winkel erzeugt wird, sondern ein *Numpy Array* das für jedes Bild im Batch einen zufälligen Winkel enthält. Im Mittel sollen die Winkel demnach bei  $0^\circ$  liegen. Der Wert für die Standardabweichung ist empirisch bestimmt, da die Werte für die Winkel nicht den tatsächlichen Winkeln in Grad entsprechen. Die Erzeugung der Winkel der Winkel  $\alpha_y$  und  $\alpha_x$  ist analog hierzu. Mit dem Unterschied, dass andere Standardabweichungen gewählt werden. Dass eine gaussche Normalverteilung verwendet wird, bedeutet, dass die meisten Piktogramme zu einer Aufnahme aus der Frontalperspektive führen sollen. 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.

Herausfinden  
was der  
Grund  
dafür ist

Die eigentliche Rotation setzt die TensorFlow Graphics Funktion `perspective_transform` um. Sie erhält einen **vierdimensionalen** Tensor an Bildern und einen **vierdimensionalen** Tensor an Rotationsmatrizen. Das bedeutet, dass der gesamte Batch an Bildern samt seiner Rotationsmatrizen übergeben wird. Somit erfolgt die Augmentation der Bilder hier parallel. Die Codezeile `transformed_imgs = 1 - transformed_imgs` kehrt hierbei 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

### 4.3.1 Laden der Datensätze

Die lokalen Pfade der Datensätze können innerhalb einer TOML-Datei konfiguriert werden.

Erklären was TOML ist und wie es geladen wird.

Es wird nun das Skript `train.py` betrachtet, das für das Training des Modells verwendet wird. Prinzipiell besitzt dieses Skript zwei Aufgaben: Es muss einerseits sowohl den Datensatz als auch die Piktogramme laden und andererseits soll es die Trainingsfunktion des Modells aufrufen.

Zunächst wird betrachtet, wie der Trainingsdatensatz des Modells geladen werden kann. Die Konfigurationsdatei enthält den Pfad der Trainingsdaten unter dem Punkt `train_data` innerhalb der Kategorie `paths`.

TensorFlow stellt 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` [31]. In folgendem Listing ist der Teil der `train.py` dargestellt, der für das Laden des Datensatzes zuständig ist.

```

1 training_path = config['paths']['train_data']
2
3 x_train = tf.keras.utils.image_dataset_from_directory(training_path,
4   ↵ batch_size=BATCH_SIZE, image_size=(IMAGE_SIZE, IMAGE_SIZE),
5   ↵ labels=None, shuffle=True, crop_to_aspect_ratio=True)
6
7 x_train_processed = utils.load_data.normalize_dataset(x_train)
```

Listing 4.4: `train.py` - Laden des Trainingsdatensatzes

An die genannte TensorFlow Funktion werden mitunter der Pfad zu den Trainingsdaten, die Batch Size und die Bildauflösung übergeben. 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*. Es kann argumentiert werden, ob nicht eine andere Interpolation besser geeignet sei für diese Studienarbeit.

Bilineare Interpolation erklären. Abbildung einfügen, die vergrößertes Bild zeigt

Das CycleGAN benötigt die Daten nicht nach ihren Klassen sortiert. Deshalb wird der Parameter `labels` auf `None` gesetzt. Das bedeutet, dass die unterschiedlichen Klassen der Daten ignoriert werden. Das kann gemacht werden, da mittels des *Cycle Consistency Losses* bestimmt werden soll, ob das Ausgangsbild von Generator X der gleichen Klasse entspricht wie sein Eingangsbild. GANs arbeiten im allgemeinen mit unüberwachtem lernen.

Durch den nächsten Parameter 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. Dies ist in der nachfolgenden Abbildung beispielhaft dargestellt:

Abbildung einfügen

Tests haben ergeben, dass die meisten Bilder des Datensatzes nur gerinfügig 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 TensorFlow *Dataset* Objekt. Eine Besonderheit hiervon ist, dass dieses Objekt seine Daten automatisch in Batches zurückgibt. Iteriert man beispielsweise mittels einer `for`-Schleife über ein *Dataset*, so ist jedes Element ein vierdimensionaler Tensor des Formats: (*Batch Größe, Breite, Höhe, Anzahl Farbkanäle*). [32]

### 4.3.2 Logging

```
1 $ tensorboard --logdir ./logs/unet  
2 $ tensorboard --logdir ./logs/resnet
```

### 4.3.3 Ergebnisse

---

Trainingsdauer pro Epoche				
Modell	Google Colab	DHBW Server	Parameter	Checkpoint Größe
UNet	30 min.	5 min.	0	1.340.240 KB
ResNet	90 min.	30 min.	0	331.709 KB

Tabelle 4.2: Vergleich von UNet und ResNet

## 4.4 Generierung

# 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**

```
1 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**

### **6.1.1 Farbhistogramme**

### **6.1.2 Wasserstein-Distanz**

### **6.1.3 Klassifizierung**

## **6.2 Evaluation der Augmentierung**

### **6.2.1 Mean Opinion Score**

## **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] I. Goodfellow, Y. Bengio und A. Courville, *Deep Learning*. MIT Press, 2016, <http://www.deeplearningbook.org>.
- [6] 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).
- [7] A. S. Glassner, „Deep Learning: A Visual Approach,“ in San Francisco: No Starch Press, 2021, ISBN: 978-1-7185-0072-3.
- [8] 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).
- [9] A. Karpathy, P. Abbeel, G. Brockman u. a., *Generative Models*, <https://openai.com/blog/generative-models/>, Letzter Zugriff: 13.01.2023, 16. Juni 2016.
- [10] 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).
- [11] A. van den Oord, N. Kalchbrenner und K. Kavukcuoglu, „Pixel Recurrent Neural Networks,“ *CoRR*, 2016. doi: [10.48550/arXiv.1601.06759](https://arxiv.org/abs/1601.06759).
- [12] M. T. Jones, *Recurrent neural networks deep dive*, <https://developer.ibm.com/articles/cc-cognitive-recurrent-neural-networks>, Letzter Zugriff: 15.01.2023, 16. Aug. 2017.
- [13] 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).

- [14] D. Bank, N. Koenigstein und R. Giryes, „Autoencoders,“ *CoRR*, 2020. doi: [10.48550/arXiv.2003.05991](https://doi.org/10.48550/arXiv.2003.05991).
- [15] 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).
- [16] 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).
- [17] 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).
- [18] 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.
- [19] 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/IVS.2019.8814090](https://doi.org/10.1109/IVS.2019.8814090).
- [20] L. Huang, *Chinese Traffic Sign Database: Traffic Sign Recognition Database*, <http://www.nlpr.ia.ac.cn/pal/trafficdata/recognition.html>, Letzter Zugriff: 25.03.2023, o.D.
- [21] United Nations Economic Commission for Europe, *Convention on Road Traffic*, 28. März 2006.
- [22] 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://doi.org/10.48550/arXiv.1909.04422).
- [23] 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).
- [24] *Road Signs Dataset*, o.D. Adresse: <https://makeml.app/datasets/road-signs>.
- [25] 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).
- [26] 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).

- [27] 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).
- [28] 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>.
- [29] TensorFlow, *CycleGAN*, <https://www.tensorflow.org/tutorials/generative/cyclegan>, Letzter Zugriff: 01.04.2023, o.D.
- [30] J. Valentin und S. Bouaziz, *Introducing TensorFlow Graphics: Computer Graphics Meets Deep Learning*, [https://www.tensorflow.org/api\\_docs/python/tf/keras/utils](https://www.tensorflow.org/api_docs/python/tf/keras/utils), Letzter Zugriff: 31.03.2023, o.D.
- [31] TensorFlow, *tf.keras.utils*, [https://www.tensorflow.org/api\\_docs/python/tf/keras/utils](https://www.tensorflow.org/api_docs/python/tf/keras/utils), Letzter Zugriff: 31.03.2023, o.D.
- [32] TensorFlow, *tf.data.Dataset*, [https://www.tensorflow.org/api\\_docs/python/tf/data/Dataset](https://www.tensorflow.org/api_docs/python/tf/data/Dataset), Letzter Zugriff: 30.03.2023, o.D.
- [33] Bundesanstalt für Straßenwesen (BASt), *Verkehrszeichen und Symbole, Verkehrszeichenkatalog 2017*, <https://www.bast.de/DE/Verkehrstechnik/Fachthemen/v1-verkehrszeichen/vz-start.html>, Letzter Zugriff: 27.03.2023, o.D.

# **Anhang**

A. Abbildungen

B. Listings

C. test

# Abbildungen



Abbildung .1: Alle 43 Piktogramme zu den generierten Klassen von Straßenschildern [33]

# 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
```