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	24.10.2022 - 08.06.2023
Matrikelnummer, Kurs	6526552, TINF20ITA
Betreuer	Prof. Dr. Monika Kochanowski

Erklärung

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

Stuttgart, 08.06.2023

Frederik Esau

Abstract

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

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

Objektivität soll sich jeder persönlichen Wertung enthalten

Kürze soll so kurz wie möglich sein

Genauigkeit soll genau die Inhalte und die Meinung der Originalarbeit wiedergeben

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

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

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

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

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

Inhaltsverzeichnis

Abkürzungsverzeichnis

CNN	Convolutional Neural Network
CycleGAN	Cycle-Consistent Generative Adversarial Network
GAN	Generative Adversarial Network
GTSRB	German Traffic Sign Recognition Benchmark
KNN	Künstliches Neuronales Netz
PixelRNN	Pixel Recurrent Neural Network

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

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 Momentane Lösungen zur Straßenschilderkennung

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. [2, S. 1ff.]

In den letzten Jahrzehnten erlebte das maschinelle Lernen und damit auch das Gebiet der KNNs einen Aufschwung. Es existiert bereits seit Mitte des vergangenen Jarhunderts, wird allerdings erst durch die zunehmende Rechenleistung und die Verfügbarkeit von großen Datenmengen flächendeckend eingesetzt. Einsatzgebiete für KNNs sind unter anderem die Objekterkennung, das Verstehen von natürlicher Sprache und die Generierung von Text und Bildern. [3, 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 sogennaten *Bias* hinzu und wenden auf das Ergebnis eine nichtlineare Funktion an. Letzere wird auch als Aktivierungsfunktion bezeichnet. Diese Aktivierungsfunktion kann analog dazu gesehen werden, dass biologische Neuronen einen Schwellenwert (*engl.: threshold*) besitzen,

der überschritten werden muss, damit das Neuron *feuert*, also einen Impuls an weitere Neuronen weitergibt. Aktivierungsfunktionen sind notwendig, damit neuronale Netze Probleme lösen können, die über die Fähigkeiten einer linearen Regression hinausgehen. Es wird eine nichtlineare Abhängigkeit zwischen dem Eingang X und dem Ausgang Y umgesetzt. [4]

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 simbolisiert. Der Ausgang (*rechts*) ist das Ergebnis der Berechnung des Neurons. [4]

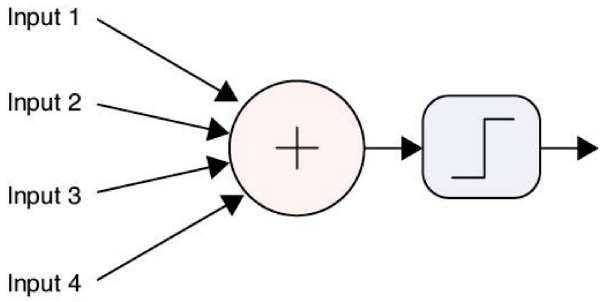

Abbildung 2.1: Einzelnes Neuron eines KNNs [4]

Um komplexe Probleme lösen zu können, werden mehre 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. [3]

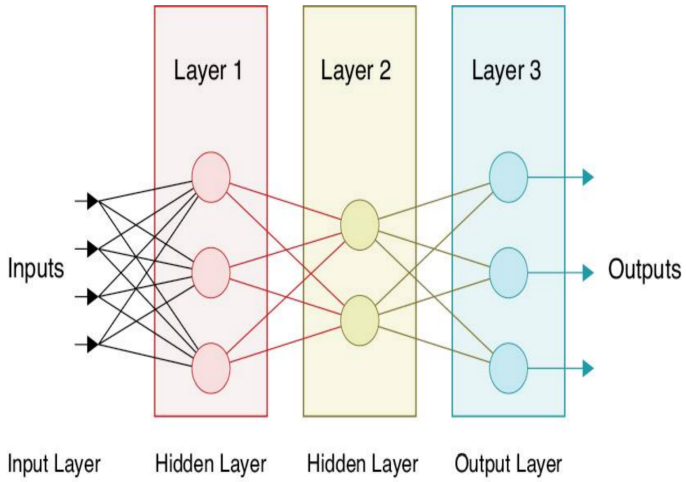

Abbildung 2.2: Vollständiges KNN [4]

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 zurfolge 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 θ kann beispielsweise folgendermaßen definiert sein:

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

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

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

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 θ. Von ihnen hängt der Wert der Kostenfunktion ab. Die Ausgangssituation ist hierbei folgende: Mit einem gegebenen θ befindet man sich in einem bestimmten Punkt der Kostenfunktion $J(\theta)$. Gesucht ist eine Parameteränderung $\mathrm{d}\theta$, 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 θ in Richtung des negativen Gradienten der Kostenfunktion bewegt. Also die Richtung, in die, aus der momentanen Ausgangsposition, die Kostenfunktion den steilsten Abstieg besitzt. Pro Trainingsiteration 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 α 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 Gradientenabtieg 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 Bildes ein Feature. Ein Netz, das Bilder der Größe 1024x1024 Pixel mit drei Farbkanälen (rot, grün blau) klassifizieren soll, muss demnach folgende Anzahl an Eingängen verarbeiten:

$$1024 \cdot 1024 \cdot 3 = 3.145.728 \tag{2.3}$$

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 Faltmatrix, 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.

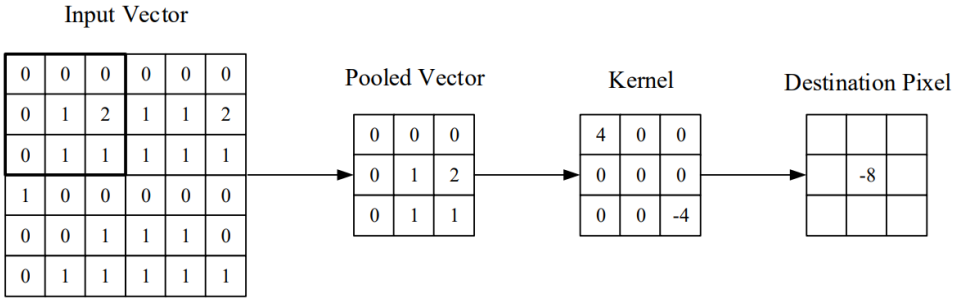

Abbildung 2.3: Beispiel für eine Faltung (engl.: Convolution) [5]

Zunächst wird der Kernel beispielsweise auf die Teilmatrix *links oben* der Eingangsmatrix angewendet. In Abbildung 2.3 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:

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

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 2.4 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.

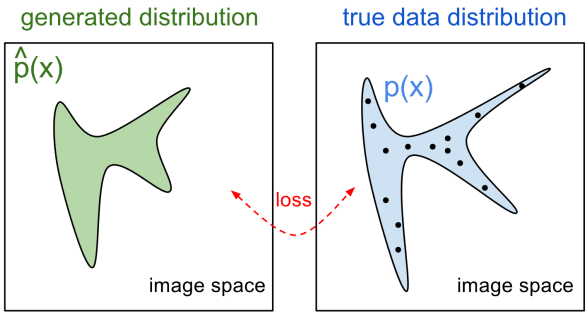

Abbildung 2.4: Beispielhafter Vergleich von $\hat{p}(x)$ und $p(x)$ [6]

Bei der Bildgenerierung versucht das Netz den Wahrscheinlichkeitswert für $\hat{p}(x)$ zu maximieren. Es erlernt durch $\hat{p}(x)$, wie die Verteilung der Trainingsdaten aussieht und versucht anschließend ausschließlich Bilder zu generieren, die dieser Verteilung folgen. Bezogen auf Abbildung 2.4 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 2.5 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.

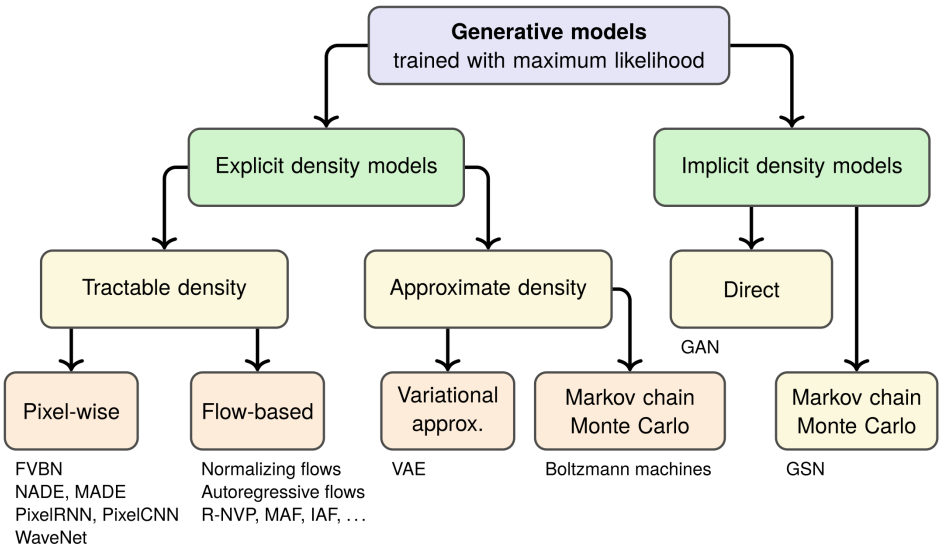

Abbildung 2.5: Taxonomie generativer Modelle [7]

Es soll in diesem Kapitel auf verschiene Architekturen generativer Netze eingegangen werden. Generative Adversarial Networks (GANs) werden gesondert in Kapitel 2.4 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. [8]

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 $(x_1, x_2, ..., x_{n^2})$ aufgeteilt werden. In Gleichung 2.5 ist dabei dargestellt, wie die Wahrscheinlichkeit eines jeden Pixels in die gesamte Verteilung $\hat{p}(x)$ einfließt. [8]

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

Jeder Pixel x_i besitzt eine eigene Wahrscheinlichkeitsverteilung $\hat{p}(x_i|x_1, ..., x_{i-1})$. Sie ist abhängig von allen anderen Pixeln $x_1, ..., x_{i-1}$ 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 $\hat{p}(x_i|x_1, ..., x_{i-1})$ 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. [8]

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 2.6 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.

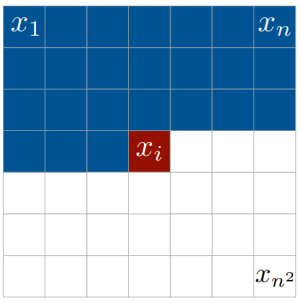

Abbildung 2.6: Bestimmung von $\hat{p}(x)$ mit PixelRNNs [8]

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 2.7 bildet hierfür eine beispielhafte Darstellung.

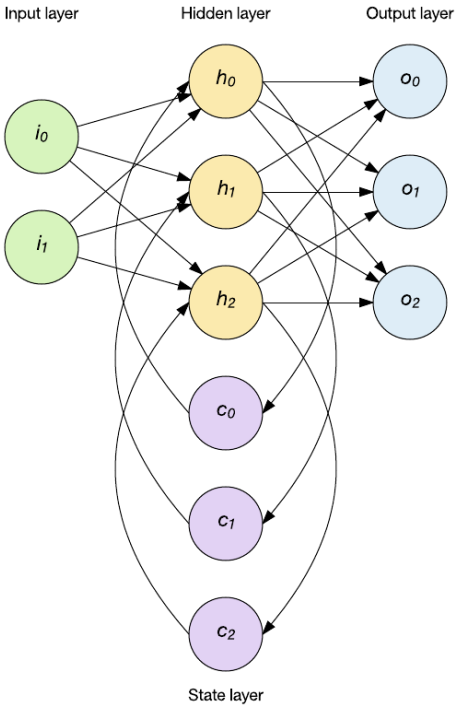

Abbildung 2.7: Darstellung eines Recurrent Neural Networks [9]

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. [10]

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. [10]

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. [8]

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 2.8 ist eine beispielhafte Autoencoderarchitektur dargestellt.

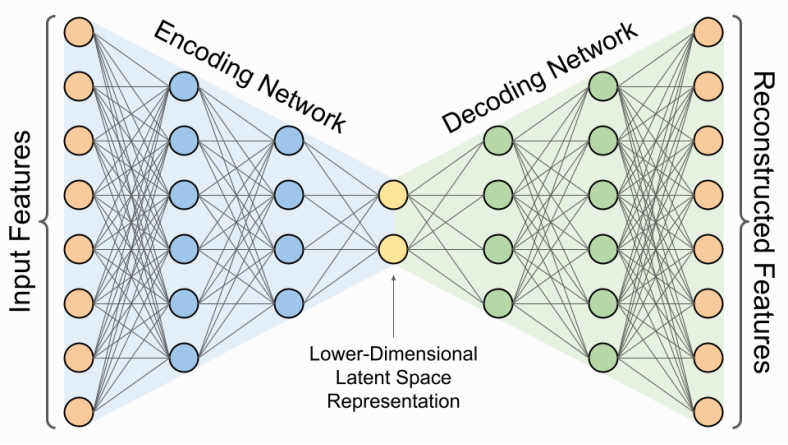

Abbildung 2.8: Architektur eines Autoencoders

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, vereinzeilte 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. [4]

Variational Autoencoder verfolgen das Ziel, eine Zufallskomponente in die Bilderzeugung 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:

$$p(z|x) \tag{2.6}$$

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 des Vektors

$\vec{z} = [z_1, z_2, ..., z_n]^\mathsf{T}$ durch eine Gaußsche Normalverteilung $\mathcal{N}(\mu_i, {\sigma_i}^2)$ beschrieben werden. Aufgabe des Kodierers ist damit nicht mehr, aus einem gegebenen x eine Menge von Merkmalen $\vec{z}$ zu bestimmen. Stattdessen soll der Kodierer die Vektoren $\vec{\mu}$ und $\vec{\sigma}$ bestimmen, durch die sich die einzelnen Normalerteilungen von $\vec{z}$ beschreiben lassen.

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

Vor- Nachteile?

[11]

2.4 Generative Adversarial Networks

GANs sind eine der eingesetzten Methoden, um Algorithmen zu entwickeln, die neue Daten künstlich generieren können. Dazu zählen zum Beispiel Bilder von Menschen, die zwar realistisch erscheinen, die so aber in der Realität nicht existieren. Die Aufgabe eines GANs ist demnach, Daten zu generieren, die denen des definierten Trainingssatzes in einer solchen Weise ähneln, dass nicht mehr unterschieden werden kann, ob ein Bild künstlich generiert ist oder aus dem Trainingssatz stammt. In dem genannten Beispiel besteht der Trainingssatz aus echten Bildern von Menschen, durch die das GAN lernt, neue Bilder zu erzeugen. Im weiteren Verlauf wird das Thema GANs ausschließlich auf die Bilderzeugung bezogen, auch wenn ebenso andere Anwendungsgebiete existieren. [4]

Ein GAN besteht aus zwei Komponenten. Dem sogenannten *Generator* und dem *Discriminator*. Der Generator erzeugt neue Bilder, während der Discriminator für jedes erzeugte Bild rät, ob es aus dem Trainingssatz stammt, oder ob es künstlich generiert ist. Nur der Discriminator hat dabei Zugriff auf den Trainingsdatensatz. Ihm werden klassischerweise abwechselnd Bilder aus dem Trainingssatz und erzeugte Bilder des Generators gezeigt. Die beiden Komponenten des GANs agieren dabei stets gegeinander. Der Generator versucht den Discriminator in die Irre zu führen, dass seine generierten Bilder in Wahrheit aus dem Trainingssatz stammen würden, während der Discriminator versucht, möglichst gut zu erkennen, ob ein Bild echt ist oder nicht. Bei beiden Komponenten handelt es sich dabei um künstliche neuronale Netze (KNNs). [4]

Während die Bildklassifizierung ein Minimierungsproblem ist, stellt die Bildgenerierung mittels GANs ein Min-Max-Problem dar. Bei ersterem wird versucht, die Loss-Function zu minimieren, sodass die Predictions möglichst gut den Labels der Daten entsprechen. Bei GANs besteht der

Unterschied darin, dass einerseits der Generator die Loss-Function zu minimieren versucht, während andererseits der Discriminator sie zu maximieren versucht.

$$\min_G \max_D V(D,G) = E_x[\log(D(x))] + E_z[\log(1 - D(G(z)))] \tag{2.7}$$

2.4.1 Spieltheorie

Aus spieltheoretischer Sicht kann das Gegenspiel von Generator und Discriminator auch als *Nullsummenspiel* betrachtet werden.

2.4.2 Training

Zu Beginn sind sowohl der Generator als auch der Discriminator nicht gut geeignet für ihre jeweiligen Aufgaben. Der Generator erzeugt Bilder, die denen des Trainingssatzes nicht ähnlich sind, während der Discriminator noch nicht weiß, was die Bilder des Trainingssatzes einzigartig macht. Durch das Training verbessern sich beide Komponenten in ihren Aufgaben. Sie trainieren sich gegenseitig, da sie beide versuchen, das Spiel zu gewinnen.

Das finale Optimum ist, dass der Discriminator so gut in der Unterscheidung zwischen echten und künstlichen Daten wird, wie mit den vorhandenen Daten nur möglich, während der Generator trotzdem in der Lage sein soll, den Discriminator zu überlisten. [4, S. 656]

Bild ist nicht 100% korrekt! G wird z.B. nicht trainiert, wenn D richtig lag, aber das Bild aus dem Datensatz stammte.

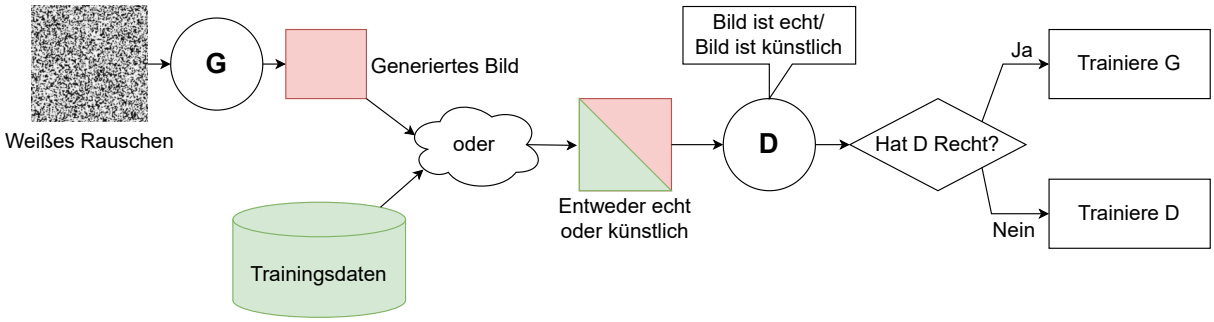

Abbildung 2.9: Trainingsprozess von GANs (angelehnt an [4, S. 654f.])

(Auf Emfindlichkeit von GANs für Hyperparameter eingehen)

2.4.3 Architekturen

Die bisher beschriebene Architektur von GANs wird auch als *Vanilla GAN* bezeichnet. Dies entspricht dem, wie GANs in Goodfellows Publikation definiert werden [12]. 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. Lösen lässt sich dieses Problem beispielsweise mit sogenannten Cycle-Consistent Generative Adversarial Networks (CycleGANs).

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 aus einem Input einen gewissen Output generieren können, sondern soll auch von einem gegebenen Output zurück auf den Input schließen können. Dies lässt sich mathematisch so darstellen, dass G und F folgende Abbildungen implementieren:

$$\mathbf{G} : X \mapsto Y \ \wedge \ \mathbf{F} : Y \mapsto X \tag{2.8}$$

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.

Was deutlich wird, ist dass hierbei die Idee verworfen werden muss, G ein weißes Rauschen als Input zuzuführen. CycleGANs sind für spezielle Anwendungsgebiete gedacht, in denen ausgewählte, und somit nicht-zufällige Eingabebilder verwendet werden. Dazu zählen Gebiete wie *Style Transfer* oder die Transformation von Bildern, respektive Bildelementen. Es wird sich im Verlauf der Arbeit zeigen, dass die Problemstellung dieser Studienarbeit ein solches Anwendungsgebiet darstellt.

$$L_{GAN}(G, D_Y, X, Y) = E_y[\log D_Y(y)] + E_x[\log(1 - D_Y(G(x)))] \tag{2.9}$$

$$L_{cyc}(G, F) = E_x[||F(G(x)) - x||_1] + E_y[||G(F(y)) - y||_1] \tag{2.10}$$

$$L(G, F, D_X, D_Y) = L_{GAN}(G, D_Y, X, Y) + L_{GAN}(F, D_X, Y, X) + \lambda L_{cyc}(G, F) \tag{2.11}$$

[13]

Wasserstein GANs [2] [4]

2.5 Vorherige Arbeiten

[14] [15]

2.6 Machine Learning Frameworks

2.6.1 Tensorflow

2.6.2 Pytorch

2.6.3 Weitere

3 Konzeption des Generative Adversarial Networks

German Traffic Sign Recognition Benchmark (GTSRB)

(a) (b) (c) (d)

Abbildung 3.1: Beispielbilder aus dem GTSRB Datensatz [16]

[16]

3.1 Datensatz

3.2 Framework

3.3 Architektur

3.4 Datenaugmentation

3.5 Training

4 Implementierung und Training

4.1 Hyperparameter

4.2 Hinzufügen äußerer Einflüsse

4.2.1 Verwackeln

4.2.2 Artefakte

4.2.3 Schnee

5 Evaluation

5.1 Metrik zur Messung der Performanz

5.2 Vergleich von Unet und Resnet

6 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.

[2] I. Goodfellow, Y. Bengio und A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[3] D. Sonnet, *Neuronale Netze Kompakt, Vom Perceptron zum Deep Learning*. Wiesbaden: Springer Vieweg Wiesbaden, 2020. DOI: 10.1007/978-3-658-29081-8.

[4] A. S. Glassner, „Deep Learning: A Visual Approach,“ in San Francisco: No Starch Press, 2021, ISBN: 978-1-7185-0072-3.

[5] K. O'Shea und R. Nash, *An Introduction to Convolutional Neural Networks*, 2015. DOI: 10.48550/ARXIV.1511.08458.

[6] A. Karpathy, P. Abbeel, G. Brockman u. a., *Generative Models*, https://openai.com/blog/generative-models/, Letzter Zugriff: 13.01.2023, 16. Juni 2016.

[7] 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.

[8] A. van den Oord, N. Kalchbrenner und K. Kavukcuoglu, „Pixel Recurrent Neural Networks,“ *CoRR*, 2016. DOI: 10.48550/arXiv.1601.06759.

[9] 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.

[10] 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.

[11] D. Bank, N. Koenigstein und R. Giryes, „Autoencoders,“ *CoRR*, 2020. DOI: 10.48550/arXiv.2003.05991.

[12] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza u. a., *Generative Adversarial Networks*, 2014. DOI: 10.48550/ARXIV.1406.2661.

[13] 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.

[14] 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.

[15] 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.

[16] J. Stallkamp et al., „The German Traffic Sign Recognition Benchmark: A multi-class classification competition," in *EEE International Joint Conference on Neural Networks*, 2011, S. 1453–1460.

Anhang

A. test

B. test

C. test