
Bachelor-Teamprojekt

Analyse, Design und Implementierung von
unterschiedlichen Generative Adversarial Network
(GAN) Architekturen im Bereich der
Bildverarbeitung

Analysis, design and implementation of different
Generative Adversarial Network (GAN)
architectures in the field of image processing

Elisa Du, Marcel Hoffmann

Mat.Nr.: 976090, 973043

Betreuer:

Prof. Dr. rer. nat. E.-G. Haffner

Datum:

01. Dezember 2023

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht. Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen. Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Ort, Datum

Unterschrift

Abstract

This is a summary of all the important points and achievements of this work.

Zusammenfassung

Hierbei handelt es sich um eine Zusammenfassung aller wichtige Punkte und Errungenschaften dieser Arbeit.

Abkürzungsverzeichnis

GAN	Generative Adversarial Network
ReLU	Rectified linear units
ResNET	Residual neural network
ML	Machine Learning
DL	Deep Learning

Inhaltsverzeichnis

1. Einleitung	1
2. Grundlagen	3
2.1. Generative Adversarial Netowrks	3
2.2. Pix2Pix	5
2.2.1. Pix2Pix-Kernkonzepte	6
2.2.2. Anwendungen von Pix2Pix	10
2.3. CycleGAN	10
2.3.1. CycleGAN - Kernkonzepte	11
2.3.2. Anwendungen von CycleGAN	14
2.4. Convolutional Layers	14
2.5. Bibliotheken	14
2.5.1. Tensorflow	14
2.5.2. Keras	15
3. Literaturreview	17
4. Problembeschreibung	19
4.1. Datenbeschaffung und -vorverarbeitung	19
4.2. Architekturentwurf	19
5. Lösungsbeschreibung	21
5.1. Training- und Testdaten	21
5.1.1. Datenladung für GAN-Training	21
5.1.2. Vorverarbeitung des Datensatzes	22
5.2. Implementierung der Pix2PixGAN-Architektur	24
5.2.1. Generator	24
5.2.2. Diskriminator	28
5.2.3. Verlustfunktion	29
5.2.4. Training und Hyperparameter	30
5.3. Implementierung der CycleGAN-Architektur	31
5.3.1. Verlustfunktion	31
5.3.2. Training und Hyperparameter	33
6. Evaluation	39
6.1. Bewertungskriterien	39
6.2. Pix2Pix: Ergebnisse und objektive Bewertung	39
6.3. CycleGAN: Ergebnisse und objektive Bewertung	39
6.4. Vergleich und Bewertung von Pix2Pix und CycleGAN	39
6.5. Vergleich von Pix2Pix und CycleGAN	39
7. Fazit und Ausblick	41
7.1. Fazit	41
7.2. Ausblick	41

A. Anhang - Code	43
B. Anhang - Dokumentationen	45
Verzeichnisse	47
Literaturverzeichnis	47
Abbildungsverzeichnis	49
Tabellenverzeichnis	51
Code-Auszugs-Verzeichnis	53
Glossar	55

1

Einleitung

Hier wird in die Arbeit eingeleitet.

2

Grundlagen

2.1. Generative Adversarial Networks

Generative Adversarial Networks, kurz GANs, sind eine aufstrebende Technologie im Bereich des maschinellen Lernens und der künstlichen Intelligenz. Inspiriert von Ian Goodfellow und seinen Kollegen im Jahr 2014, bieten GANs eine effiziente Möglichkeit, tiefe Repräsentationen von Daten zu erlernen, ohne dass große Mengen an annotierten Trainingsdaten benötigt werden(CWD⁺18). Dies wird durch die Verwendung von Backpropagation und den Wettbewerb zwischen zwei neuronalen Netzen - dem Generator und dem Diskriminator - erreicht. Daraus ergeben sich zahlreiche neue Ansätze zur Generierung realistischer Inhalte. Die Anwendungen reichen von der Bildgenerierung bis hin zur Superauflösung und Textgenerierung(AMB21).

Funktionsweise

Der Generator und der Diskriminator sind die Hauptkomponenten eines GAN. Die beiden neuronalen Netze werden gleichzeitig trainiert und konkurrieren miteinander, wobei der Generator versucht, den Diskriminator zu täuschen, indem er synthetische Inhalte erzeugt. Um die Glaubwürdigkeit des Generators zu erhöhen, so dass der Diskriminator nicht mehr zwischen den Eingaben unterscheiden kann, wird das gesamte Netz trainiert. Die Netze werden in der Regel als mehrschichtige Netzwerke implementiert, die aus Convolutional und Fully-Connected Schichten bestehen(CWD⁺18).

Generator

Der Generator dient zur Erzeugung künstlicher Daten wie Bilder und Texte. Der Generator ist nicht mit dem realen Datensatz verbunden und lernt daher nur durch die Interaktion mit dem Diskriminator. Wenn der Diskriminator nur noch 50% der Eingaben richtig vorhersagt, gilt der Generator als optimal(CWD⁺18).

Diskriminator

Die Unterscheidung zwischen echten und unechten Eingaben ist Aufgabe des Diskriminators. Der Diskriminator kann sowohl künstliche als auch reale Daten ver-

wenden. Wenn der Diskriminator nicht mehr richtig unterscheiden kann, wird er als konvergierend bezeichnet(AMB21). Andernfalls wird er als optimal bezeichnet, wenn seine Klassifizierungsgenauigkeit maximiert ist. Im Falle eines optimalen Diskriminators wird das Training des Diskriminators gestoppt und der Generator trainiert alleine weiter, um die Genauigkeit des Diskriminators wieder zu verbessern(CWD+18).

Training

Durch das Finden von Parametern für beide Netze wird das Training durchgeführt. Ziel ist die Optimierung beider Netze durch Anwendung von Backpropagation zur Verbesserung der Parameter. Das Training wird oft als schwierig und instabil beschrieben. Einerseits scheint es eine Herausforderung zu sein, beide Modelle zur Konvergenz zu bringen. Andererseits kann der Generator sehr ähnliche Muster für verschiedene Eingaben erzeugen, was als "Mode-Collapse Problem" bezeichnet wird. Der Diskriminatorverlust kann schnell gegen Null konvergieren, so dass es keinen zuverlässigen Weg gibt, den Gradienten zum Generator zu aktualisieren. Zur Lösung dieser Probleme wurden verschiedene Ansätze vorgeschlagen, wie z.B. die Verwendung heuristischer Verlustfunktionen. Eine weitere Möglichkeit, die von Sonderby et al. vorgeschlagen wurde, besteht darin, den Datensatz vor der Verwendung zu verrauschen(CWD+18).

Adversarial Loss

Der Erfolg von GAN liegt zum Einen an dem verwendeten Adverarsial Verlustfunktion. Dieser

// TODO: PAPER ABOUT THAT

Anwendungen

GAN wurde ursprünglich für unüberwachtes maschinelles Lernen entwickelt. Die Architektur liefert jedoch ebenso gute Ergebnisse beim halbüberwachten Lernen und beim Reinforcement Learning(AMB21). Aus diesem Grund wird sie in einer Vielzahl von Bereichen wie dem Gesundheitswesen, dem Maschinenbau und dem Bankwesen eingesetzt. Beispielsweise wird GAN in der Medizin zur Erkennung und Behandlung chronischer Krankheiten eingesetzt. Aber auch die Identifikation von 3D-Objekten und die Generierung von realen Bildern und Texten ist durch den Einsatz von GANs möglich.

Limitationen

Ein kritisches Problem von GANs ist die Instabilität des Trainings aufgrund von Mode-Collapse, was die Weiterentwicklung des generativen Lernens und potentielle Anwendungen einschränkt(LLW+). Der Generator lernt nur Bilder bestimmter

Arten der Datenverteilung. Andere Arten, die ebenfalls in der Verteilung vorkommen, werden hingegen vernachlässigt([SVR⁺](#)). Ansätze wie das Hinzufügen von Rauschen zum Netzwerk, eine Manifold Entropy Estimation ([LLW⁺](#)) und implizites Variationslernen ([SVR⁺](#)) wurden bereits vorgeschlagen, um dieses Problem zu lösen.

Des Weiteren birgt die Fähigkeit eines GANs zur Generierung von Inhalten, die nahezu identisch mit authentischen Inhalten sind, potenzielle Herausforderungen in realen Szenarien, insbesondere im Zusammenhang mit der menschlichen Bildsynthese. Diese Fähigkeit ermöglicht es Betrügern, gefälschte Profile in sozialen Medien zu erstellen. Gezielte Anwendungen von GANs, die darauf ausgelegt sind, einzigartige und realistische Bilder von Personen zu erzeugen, die in der Realität nicht existieren, könnten die Erstellung falscher Profile erschweren([AMB21](#)).

2.2. Pix2Pix

Isola et al., hat sich als zentrales Framework für Bild-zu-Bild-Übersetzungen auf der Basis von bedingten generativen adversariellen Netzwerken (cGANs) etabliert. Es ermöglicht die Erstellung einer abstrakten Abbildung von einem Eingangsbild zu einem korrespondierenden Ausgangsbild und bewältigt dabei eine vielfältige Palette an Bildübersetzungsaufgaben, wie die Transformation von Skizzen in realistische Bilder oder die Konvertierung von Tages- zu Nachtaufnahmen.

Pix2Pix fungiert hier als Generative Adversarial Network (GAN), spezialisiert auf diverse Formen der Bildübersetzung. Darunter fallen die Umwandlung von Schwarz-Weiß-Fotos in Farbbilder, die Transformation von Skizzen in realistische Bilder, und relevant für diese Arbeit, die Konvertierung von Satellitenbildern in kartographische Darstellungen, ähnlich den Visualisierungen von Google Maps.

Die Architektur von Pix2Pix besteht aus einem Generator und einem Diskriminator. Der Generator, der eine U-Net-Architektur verwendet, besteht aus einem Encoder und einem Decoder. Der Encoder komprimiert das Eingangsbild schrittweise zu einer niedrigdimensionalen Repräsentation, während der Decoder diese dazu nutzt, das Ausgangsbild zu rekonstruieren. Skip-Verbindungen zwischen Encoder und Decoder helfen dabei, sowohl globale als auch lokale Informationen im generierten Bild zu bewahren.

Der Diskriminator nimmt die Form eines PatchGAN-Modells an und bewertet Patches des Bildes, indem er die Wahrscheinlichkeit für die Echtheit jedes Patches ausgibt. Dies ermöglicht die Anwendung des Diskriminators auf Bilder unterschiedlicher Größen. Im Zuge des adversariellen Trainingsprozesses passen sowohl der Generator als auch der Diskriminator ihre Fähigkeiten fortlaufend an. Während der Generator lernt, immer realistischere Übersetzungen zu erzeugen, wird der Diskriminator stetig besser darin, zwischen echten und generierten Bildern zu unterscheiden.

2.2.1. Pix2Pix-Kernkonzepte

Generator

Die Bildverarbeitung hat in den letzten Jahren durch den Einsatz tiefer neuronaler Netzwerke erhebliche Fortschritte gemacht. Im Mittelpunkt vieler dieser Fortschritte steht die U-Net-Architektur, die speziell für die Bildsegmentierung entwickelt wurde. Diese Architektur zeichnet sich durch ihre angeklügelte Kombination aus Encoder- und Decoder-Strukturen sowie durch den Einsatz von Skip-Verbindungen aus (PJTA).

Bei der Encoder-Decoder-Struktur handelt es sich um einen Ansatz, bei dem das Eingangsbild zunächst durch den Encoder schrittweise reduziert wird. Dieser Prozess dient dazu, wesentliche Merkmale des Bildes zu erfassen. Anschließend wird das Bild durch den Decoder wiederhergestellt, indem die zuvor extrahierten Merkmale verwendet werden. Während dieser Prozesse besteht jedoch das Risiko des Informationsverlustes, insbesondere in den tieferen Schichten des Netzwerks. Um dieses Problem zu adressieren, führt die U-Net-Architektur Skip-Verbindungen ein. Diese direkten Verbindungen zwischen korrespondierenden Schichten des Encoders und Decoders sorgen dafür, dass Detailinformationen nicht verloren gehen. Genauer gesagt, ermöglichen diese Verbindungen den direkten Informationsfluss zwischen jeweils äquivalenten Schichten, wodurch die Rekonstruktion des Bildes im Decoder mit einer höheren Genauigkeit erfolgt (PJTA).

Die Bedeutung von Skip-Verbindungen zeigt sich insbesondere in Anwendungen wie der Bild-zu-Bild-Übersetzung. Hier muss oft ein Bild mit niedriger Auflösung in ein Bild mit hoher Auflösung überführt werden, ohne dass Details verloren gehen. Die U-Net-Architektur, die angereichert mit diesen Verbindungen ist, ermöglicht daher eine feinere Rekonstruktion, die sowohl globale als auch lokale Informationen berücksichtigt (PJTA).

Somit kann die U-Net-Architektur durch ihre Kombination aus Encoder-Decoder-Struktur und Skip-Verbindungen ein effektives Werkzeug für die Bildsegmentierung darstellen. Ihre Fähigkeit, sowohl globale Muster als auch feine Details zu berücksichtigen, macht sie zu einer bevorzugten Wahl für viele Bildverarbeitungsaufgaben (PJTA).

In Abbildung 2.1 ist die typische U-Net-Architektur dargestellt. Die linke Seite des "U" repräsentiert den Encoder-Teil, der das Eingangsbild schrittweise reduziert und wesentliche Merkmale extrahiert. Die rechte Seite repräsentiert den Decoder-Teil, der das Bild mithilfe der extrahierten Merkmale rekonstruiert. Die horizontalen Linien repräsentieren die Skip-Verbindungen, die sicherstellen, dass Detailinformationen zwischen den korrespondierenden Schichten des Encoder und Decoders direkt übertragen werden (PJTA).

In der Pix2Pix Technologie dient diese U-Net-Architektur als Generator. Er ist das zentrale Element, das für die Bild-zu-Bild-Übersetzung verantwortlich ist. Die Wahl der U-Net-Struktur für den Generator liegt in ihrer Fähigkeit, feinere Details und Kontextinformationen aus dem Eingangsbild beizubehalten, was für die Bild-zu-Bild-Übersetzung von entscheidender Bedeutung ist. Die Encoder-

Decoder-Struktur des U-Net ermöglicht es dem Generator, den globalen Kontext des Bildes zu erfassen, während die Skip-Verbindungen sicherstellen, dass auch lokale Details im resultierenden Bild berücksichtigt werden (PJTA).

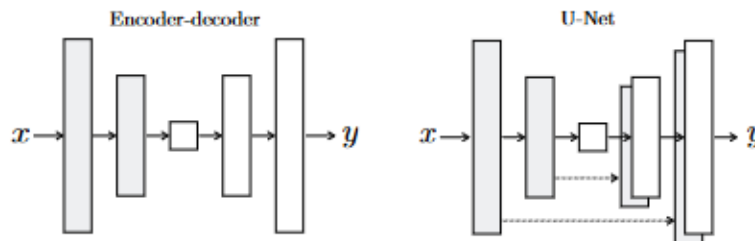


Abbildung 2.1.: Schematische Darstellung der U-Net-Architektur. Die Architektur besteht aus einem Encoder-Teil (links), einem Decoder-Teil (rechts) und Skip-Verbindungen zwischen korrespondierenden Schichten.

Diskriminator

Im adversariellen Lernprozess spielen Generatoren und Diskriminatoren eine entscheidende Rolle. Während der Generator versucht, Daten zu erzeugen, die von echten Daten kaum zu unterscheiden sind, evaluiert der Diskriminator die vom Generator erzeugten Daten und versucht, zwischen echten und gefälschten Daten zu unterscheiden (PJTA).

Im Kontext von Generative Adversarial Networks (GANs), insbesondere im speziellen Fall des Pix2Pix GANs, spielt der PatchGAN-Diskriminator eine besonders wichtige Rolle. Der zentrale Unterschied dieses Diskriminators zu traditionellen Diskriminatoren liegt in der Art und Weise, wie er Bilder bewertet. Statt das gesamte Bild zu beurteilen, zerlegt der PatchGAN-Diskriminator das Bild in mehrere kleinere Bildabschnitte oder Patches und bewertet jeden Patch einzeln auf seine Echtheit (PJTA).

Ein solches Vorgehen hat den klaren Vorteil, dass feinere Strukturen und Details in den Bildern erkannt und beurteilt werden können. Durch diese segmentierte Beurteilung kann der Diskriminator besser einschätzen, ob die Struktur und Beschaffenheit eines bestimmten Bildteils realistisch ist. Dies ist besonders nützlich, da kleinere Unstimmigkeiten in den Bildern, die ein traditioneller Diskriminator möglicherweise übersieht, vom PatchGAN erfasst werden können.

Ein weiterer Vorteil des PatchGAN-Diskriminators ist seine Skalierbarkeit. Da er auf festen Patchgrößen basiert, kann er flexibel auf Bilder unterschiedlicher Größen angewendet werden, ohne dass das zugrunde liegende Modell geändert werden muss. Dies führt nicht nur zu einer schnelleren Bildverarbeitung, sondern ermöglicht auch eine effiziente Ausführung auf großen Bildern. Darüber hinaus reduziert es potenzielle Kachelartefakte, die bei traditionellen Diskriminatoren auftreten können (PJTA).

Eine Metrik, die oft verwendet wird, um die Leistung des Diskriminators zu beurteilen, ist der FCN-Score. Dieser bewertet die Qualität der vom Generator erzeugten

Bilder. Ein hoher FCN-Score zeigt, dass der Diskriminator erfolgreich echte von gefälschten Bildern unterscheiden kann.

Der PatchGAN-Diskriminator kann wenn er effektiv eingesetzt wird, zu besseren und realistischeren Bildern im adversariellen Lernprozess beitragen. Seine Fähigkeit, lokale Bildinformationen zu bewerten, ermöglicht es auch subtile Unterschiede in den Bildern zu erkennen, was zu einer verbesserten Qualität der generierten Bilder führt (PJTA).

1

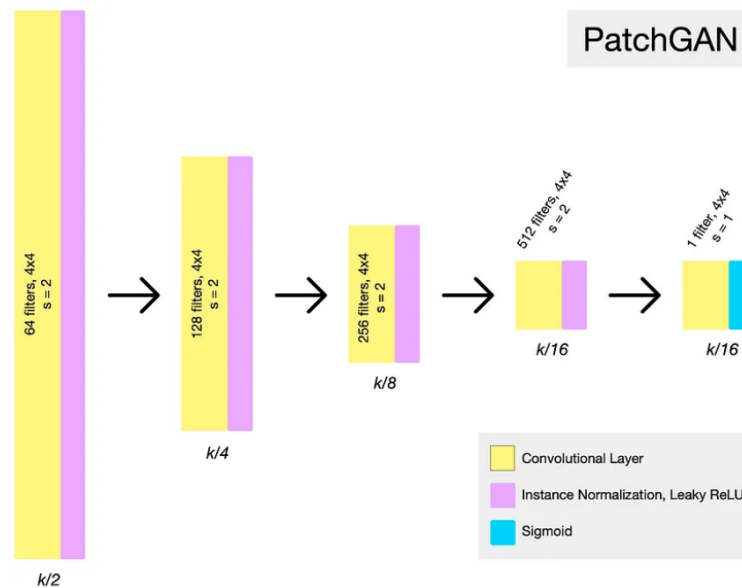


Abbildung 2.2.: Eine mögliche Architektur eines PatchGAN Diskriminator¹

L1-Verlustfunktion

Die L1-Verlustfunktion, auch bekannt als Mean Absolute Error (MAE), spielt eine entscheidende Rolle im Pix2Pix-Modell, einem bedingten Generative Adversarial Network (cGAN) für Bild-zu-Bild-Übersetzungen. Diese Verlustfunktion misst den durchschnittlichen absoluten Unterschied zwischen den vorhergesagten und den tatsächlichen Werten, wodurch sie die Genauigkeit der generierten Bilder verbessert, insbesondere im Hinblick auf die niedrigen Frequenzen im Bild. Die L1-Verlustfunktion trägt somit maßgeblich zur Bewahrung der strukturellen Integrität und des Kontexts des Bildes bei (PJTA).

Die Verwendung des L1-Verlusts zusätzlich zum adversariellen Verlust im Pix2Pix-Modell ist von entscheidender Bedeutung. Während der adversarielle Verlust darauf abzielt, die generierten Bilder realistisch erscheinen zu lassen, konzentriert sich der L1-Verlust auf die Genauigkeit der niedrigen Frequenzen, um die strukturelle

¹<https://towardsdatascience.com/cyclegan-learning-to-translate-images-without-paired-training-data-a5b4e93862c8d>

Integrität und den Kontext des Bildes zu bewahren. Diese Kombination ermöglicht es, sowohl die niedrigen als auch die hohen Frequenzen im Bild effektiv zu erfassen, was zu generierten Bildern führt, die sowohl strukturell korrekt als auch visuell ansprechend sind (PJTA).

Die L1-Verlustfunktion neigt jedoch dazu, bei den hohen Frequenzen unscharfe Ergebnisse zu liefern. Dies liegt daran, dass der L1-Verlust den Median der möglichen Werte bevorzugt, was zu einer Glättung der Bildtexturen führen kann. Um dieses Problem zu adressieren und scharfe, hochfrequente Details im Bild zu erhalten, wird der L1-Verlust im Pix2Pix-Modell mit einem adversariellen Verlust kombiniert. Diese synergetische Kombination von Verlustfunktionen ermöglicht es dem Pix2Pix-Modell, hochwertige Bild-zu-Bild-Übersetzungen durchzuführen, die sowohl visuell ansprechend als auch strukturell korrekt sind (PJTA).

Darüber hinaus hat sich die Kombination von L1-Verlust und adversariellen Verlust im Pix2Pix-Modell als nützlich für eine Vielzahl von Bild-zu-Bild-Übersetzungsproblemen erwiesen, einschließlich semantischer Segmentierung und Farbgebung. Durch die effektive Erfassung sowohl der niedrigen als auch der hohen Frequenzen im Bild trägt das Pix2Pix-Modell dazu bei, die Qualität der generierten Bilder zu verbessern und ihre Anwendbarkeit auf verschiedene Probleme zu erweitern (PJTA).

Training

Der Trainingsprozess von Pix2Pix-Generative Adversarial Networks in der Bild-zu-Bild-Übersetzung geht über die bloße Erlernung der Abbildung von Eingabe- zu Ausgabebildern hinaus. Er umfasst auch das Entwickeln einer maßgeschneiderten Verlustfunktion, die speziell auf diese Art der Bildtransformation abgestimmt ist. Dieser umfassende Ansatz ermöglicht es Pix2Pix, sich flexibel an eine Vielzahl von Problemen anzupassen, die in der Vergangenheit unterschiedliche und spezialisierte Ansätze für die Verlustfunktion erforderten. Dadurch wird die breite Anwendbarkeit und Effektivität des Pix2Pix-Modells in verschiedenen Bildübersetzungsaufgaben deutlich. Pix2Pix benötigt eine spezifische Art von Trainingsdaten, um effektiv zu funktionieren. Die Trainingsdaten bestehen aus Paaren von Bildern, wobei jedes Paar ein Eingabebild und das entsprechende Ausgabebild enthält. Diese Bilder können 1-3 Kanäle aufweisen, was bedeutet, dass das Modell sowohl mit monochromatischen (Graustufen) als auch mit farbigen Bildern (RGB) arbeiten kann. Diese Flexibilität in der Kanalverarbeitung ermöglicht eine breitere Anwendung des Pix2Pix-Modells auf verschiedene Bildtypen. Die Bilder müssen nicht in einer bestimmten Weise vorverarbeitet werden, da das Modell direkt auf den Rohpixeln arbeitet. Diese Flexibilität in der Kanalverarbeitung und die Fähigkeit, direkt auf Rohpixeln zu arbeiten, unterstreichen die Vielseitigkeit des Pix2Pix-Modells. Dies wird weiter durch die sorgfältige Auswahl spezifischer Trainingsparameter und Hyperparameter-Optimierungsstrategien ergänzt. Der Adam-Gradientenoptimierungsalgorithmus ist eine Methode zur Optimierung des maschinellen Lernens, die auf adaptiven Schätzungen niedrigerer Ordnung basiert. Er passt automatisch die Lernrate während des Trainings an und eignet sich besonders gut für Probleme mit großen Datensätzen und/oder vielen Parametern.

Durch empirische Tests hat sich herausgestellt, dass eine initiale Lernrate von 0.0002 und die Momentum-Parameter von $\beta_1 = 0.5$ und $\beta = 0.999$ optimal sind, um die Balance zwischen Lerngeschwindigkeit und Stabilität des Trainingsprozesses zu optimieren. Auch die Wahl einer kleinen Batchgröße, typischerweise 1, spielt eine entscheidende Rolle, um die Trainingseffizienz zu maximieren und qualitativ hochwertige Ergebnisse zu erzielen. Diese spezifischen Einstellungen der Trainingsparameter tragen maßgeblich dazu bei, das Potenzial des Pix2Pix-Modells voll auszuschöpfen.(PJTA).

Im Rahmen des Trainingsprozesses von Pix2Pix wird eine iterative Methode verwendet, bei der der Generator und der Diskriminator abwechselnd trainiert werden. Zunächst werden die Trainingsdaten vorbereitet, die aus Paaren von Eingabe- und Zielbildern besteht. Diese Bilder werden auf einen Wertebereich von -1 bis +1 skaliert. Diese Normalisierung ist wichtig, da sie zur Stabilisierung des Trainingsprozesses beiträgt. Sie ermöglicht es dem Modell, mit einem standardisierten Datensatz zu arbeiten, was die Lernrate und die Konvergenzgeschwindigkeit verbessert. Während des Trainings versucht der Generator, Bilder zu erzeugen, die vom Diskriminator nicht von realen Bildern unterschieden werden können. Der Diskriminator hingegen lernt, zwischen echten und vom Generator erzeugten Bildern zu unterscheiden. Eine Schlüsselkomponente dieses Prozesses ist die Verwendung einer zusammengesetzten Verlustfunktion, die sowohl den adversariellen Verlust (bewertet vom Diskriminator) als auch den L1-Verlust (mittlerer absoluter Fehler zwischen generiertem Bild und Zielbild) umfasst. Dadurch wird der Generator dazu angehalten, realistische Übersetzungen der Eingabebilder zu generieren. Dieses Gleichgewicht zwischen Generator und Diskriminator ist entscheidend für die Effektivität des Pix2Pix-Modells (Haz21).

2.2.2. Anwendungen von Pix2Pix

Hier können Sie über die Anwendungen von Pix2Pix schreiben.

2.3. CycleGAN

CycleGAN, das 2017 von Jun-Yan Zhu et al. vorgestellt wurde, stellt eine neue Entwicklung im Bereich des maschinellen Lernens und insbesondere der Bildübersetzung zwischen unpaaren Domänen dar. Es erweitert die Pix2Pix-Architektur durch die Einführung einer Cycle Consistency Loss-Funktion, die sicherstellt, dass das Originalbild nach einem Übersetzungs- und Rückübersetzungszyklus erhalten bleibt. Der Generator G transformiert Bilder aus der Domäne X in die Domäne Y , während der Generator F den umgekehrten Prozess durchführt. Diese Transformationen werden ohne gepaarte Trainingsdaten durchgeführt.

2.3.1. CycleGAN - Kernkonzepte

Architektur der Generatoren

Die Architektur der Generatoren in CycleGAN spielt eine entscheidende Rolle bei der erfolgreichen Durchführung von Bildübersetzungen zwischen verschiedenen Domänen. Typischerweise basieren die Generatoren auf dem ResNet-Ansatz, der für seine Fähigkeit bekannt ist, tiefe neuronale Netze zu trainieren (HZRS).

¹

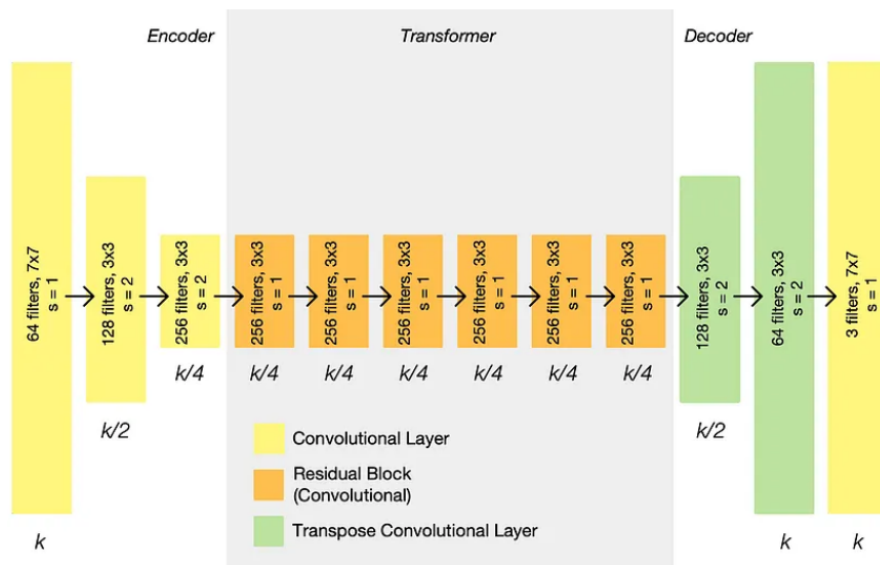


Abbildung 2.3.: Eine Architektur eines CycleGAN Generators. Instanznormalisierung und ReLU Aktivierung erfolgt nach jeder Schicht ¹

Im Rahmen der Architektur von Zhu et al. manifestiert sich der Generator im CycleGAN in drei zentralen Abschnitten, wie graphisch in Abbildung 2.3 illustriert. Der Encoder besteht aus drei Convolutional-Schichten, welche unmittelbar auf das Eingabebild einwirken und dabei die Repräsentationsgröße reduzieren sowie die Kanalanzahl erhöhen. Das resultierende Bild unterzieht sich einem Transformer, zusammengesetzt aus mehreren Residualblöcken. Die aus dieser Transformation hervorgehende Repräsentation durchläuft den Decoder, welcher aus zwei Transpose Convolutional-Schichten besteht und somit das Bild erneut vergrößert. Die finale RGB-Ausgabe wird durch eine Ausgabeschicht generiert. Jede dieser Schichten ist mit Instanznormalisierung und ReLU-Aktivierung versehen, was sowohl die Trainingsstabilität fördert, als auch die Qualität der generierten Bilder optimiert (UVL, NH10).

Die ResNet-Methode, von Kaiming He et al. im Jahr 2015 eingeführt, bietet eine Lösung für das Degradationsproblem. Dieses Phänomen tritt auf, wenn tie-

¹<https://towardsdatascience.com/cyclegan-learning-to-translate-images-without-paired-training-data-5b4e93862c8d>

fe neuronale Netze bei Zugabe zusätzlicher Schichten schlechtere Leistungen erbringen als flachere Netze, da die Rückwärtspropagierung von Fehlern in tiefen Netzwerken erschwert wird. Die Integration von Residualblöcken ermöglicht die Überwindung dieses Problems durch die Hinzufügung einer Identitätsabbildung. Das Netzwerk lernt diese Abbildung, indem es das Residuum auf Null setzt. Residualblöcke dienen dazu, Änderungen und Fehler zu erlernen, die notwendig sind, um von der Eingabe zur gewünschten Ausgabe zu gelangen. Dies wird durch Shortcut-Verbindungen realisiert, die eine oder mehrere Ebenen überspringen und am Ende einer gestapelten Schicht hinzugefügt werden. Solche Verbindungen fügen keine zusätzlichen Parameter oder Rechenleistung hinzu, und das gesamte Netzwerk kann weiterhin mittels stochastischem Gradientenabstieg (SDG) trainiert werden (HZRS).

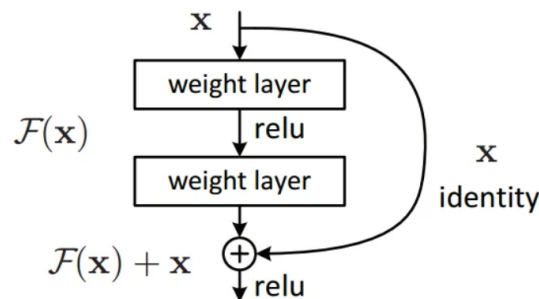


Abbildung 2.4.: Ein Aufbau eines Residualblocks

Architektur der Diskriminatoren

In Pix2Pix ist die gängige Architektur für Diskriminatoren ein PatchGAN, bei dem das Bild in kleine Patches unterteilt wird und jeder Patch separat klassifiziert wird (vgl. Abschnitt 2.2.1). Diese Vorgehensweise ermöglicht eine präzise Unterscheidung zwischen echten und generierten Bildern auf lokaler Ebene, was besonders in Bezug auf die feinstrukturierte Bewertung von Bildabschnitten von Vorteil ist (ZPIE).

In den Implementierungen von CycleGAN wird im Unterschied zu Pix2Pix auf die Verwendung von Batch Normalisierung verzichtet, und stattdessen wird Instanznormalisierung bevorzugt. Bei der Instanznormalisierung wird jedes Bild individuell betrachtet, ohne Berücksichtigung über die gesamte Batch-Dimension hinweg. Dieser Ansatz bietet eine effektivere Stilübertragung im Feed-Forward-Modus und weist eine schnellere Konvergenz auf im Vergleich zur Batch Normalisierung (HB). Eine mögliche Architektur ist in Abbildung 2.2 veranschaulicht.

Training

Das Training von CycleGAN erfolgt nach einem kompetitiven Verfahren. Die Generatoren $G : X \rightarrow Y$ und $F : Y \rightarrow X$ konkurrieren mit den entsprechenden Diskriminatoren D_X und D_Y . D_X versucht, die von F erzeugten Bilder von den

echten Bildern aus X zu unterscheiden, während D_Y versucht, die von G erzeugten Bilder von den echten Bildern aus der Domäne Y zu unterscheiden. Die adversen Verluste sind so optimiert, dass die erzeugten Bilder für die Diskriminatoren kaum von den echten Bildern zu unterscheiden sind. (ZPIE).

Cycle - Konsistenz

CycleGAN führt zusätzlich eine Cycle-Konsistenz ein. Diese stellt sicher, dass die Übersetzungen zwischen den Domänen sowohl vorwärts (X nach Y) als auch rückwärts (Y nach X) konsistent sind.

Die Kernidee besteht darin, dass nach der Übersetzung von X nach Y und zurück nach X das resultierende Bild dem ursprünglichen X entsprechen sollte. Um dies zu erreichen, wird die Differenz zwischen dem Originalbild x und dem zyklisch übersetzten Bild $F(G(x))$ mit Hilfe der L1-Verlust minimiert.

Durch die Einführung dieser zyklischen Konsistenz wird das Problem des Modekollapses gelöst. Die weitere Verlustfunktion stellt sicher, dass die generierten Bilder mehr Strukturen enthalten und somit konsistentere Übersetzungen zwischen den Domänen liefern (ZPIE).

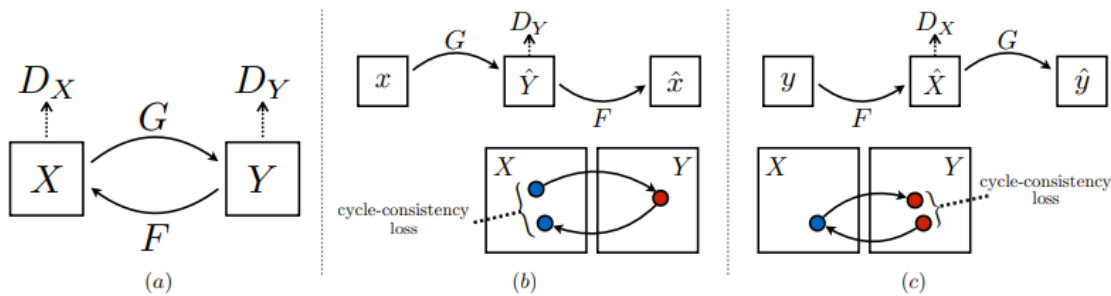


Abbildung 2.5.: (a) Modell des CycleGANs, bestehend aus zwei Generatoren $F : Y \rightarrow X$ und $G : X \rightarrow Y$ und zugehörige adversarielle Diskriminatoren D_X und D_Y , (b) Cycle-Konsistenz $F(G(x)) \approx x$, (c) Cycle-Konsistenz $G(F(y)) \approx y$ (ZPIE)

Identity - Loss

Zusätzlich zu den adversariellen und zyklischen Verlusten kann ein Identitätsverlust in die Gesamtverlustfunktion integriert werden, um sicherzustellen, dass die Farbkomposition des Eingabebildes beibehalten wird, während es ins Ausgabebild übersetzt wird. Insbesondere bei der Erstellung von Fotografien aus Gemälden hat sich diese Methode bewährt. Wenn der Generator G ein Bild aus dem Bereich Y erhält, darf es sich aufgrund seiner bereits vorhandenen Zugehörigkeit zu diesem Bereich nicht mehr verändern. Der Verlust wird dabei mittels des L1-Verlust ermittelt, bei dem die Differenz zwischen den Pixeln des generierten Bildes $G(y)$ und dem Referenzbild $y \in Y$ erfasst wird. Das gleiche Verfahren wird für den anderen Generator F angewendet (ZPIE).

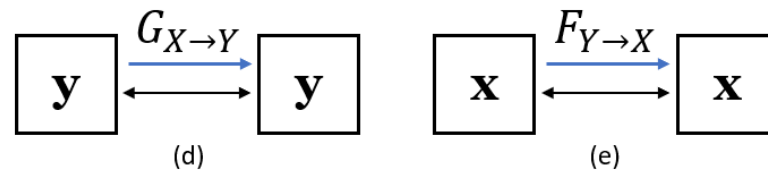


Abbildung 2.6.: Identity-Mapping für (d) Generator G und (e) Generator F

2.3.2. Anwendungen von CycleGAN

CycleGAN hat sich als äußerst vielseitiges Modell erwiesen und wird in verschiedenen Anwendungsbereichen eingesetzt. Die Fähigkeit, Bildübersetzungen zwischen unpaaren Domänen durchzuführen, hat zu zahlreichen innovativen Anwendungen geführt.

Eine der prominentesten Anwendungen von CycleGAN ist die Bild-zu-Bild-Übersetzung. Dies beinhaltet die Transformation von Bildern zwischen verschiedenen Stilen, Szenarien oder Kunstwerken. Beispielsweise kann CycleGAN verwendet werden, um Fotos in den Stil berühmter Kunstwerke zu transformieren, was einen einzigartigen und kreativen Ansatz für die Bildbearbeitung bietet. Ein weiteres Anwendungsgebiet von CycleGAN ist die Stilübertragung. Hier können Stile von einem Bild auf ein anderes übertragen werden, ohne dass gepaarte Trainingsdaten benötigt werden. So ist es möglich, den Stil eines Gemäldes auf ein fotografisches Bild zu übertragen oder umgekehrt. CycleGAN kann auch für die Übersetzung zwischen verschiedenen Farbbereichen verwendet werden. Zum Beispiel kann es verwendet werden, um Schwarz-Weiß-Bilder in Farbversionen zu übersetzen oder den Farbtönen von Bildern anzupassen, ohne dass gepaarte Trainingsdaten benötigt werden.

2.4. Convolutional Layers

// TODO

2.5. Bibliotheken

2.5.1. Tensorflow

TensorFlow ist ein weit verbreitetes Open-Source-Framework, das für maschinelles Lernen und tiefe neuronale Netzwerke eingesetzt wird¹. Es bietet eine umfangreiche Plattform zur Entwicklung und Umsetzung von Modellen in verschiedenen Anwendungsbereichen, darunter Bilderkennung und natürliche Sprachverarbeitung. Die Architektur von TensorFlow ermöglicht das Erstellen komplexer maschineller Lernanwendungen und stellt umfassende Tools und Bibliotheken zur Verfügung, um den gesamten Entwicklungsprozess zu unterstützen. Darüber hinaus profitiert

¹<https://www.tensorflow.org/>

TensorFlow von einer aktiven und engagierten Community, die kontinuierlich zur Weiterentwicklung des Frameworks beiträgt².

2.5.2. Keras

Ursprünglich als eigenständiges Framework konzipiert und seit TensorFlow 2.0 in die TensorFlow Core API integriert, fungiert Keras als Open-Source-API zur Modellierung von Strukturen im Bereich des Deep Learning³. Diese Bibliothek, die in der Programmiersprache Python implementiert ist, umfasst sämtliche Phasen des maschinellen Lern-Workflows. Beginnend mit der Datenverarbeitung ermöglicht sie eine nahtlose Fortführung bis zur präzisen Abstimmung der Hyperparameter während des Trainingsprozesses. Die Grundsätze von Keras sind durch ihre Einfachheit, Flexibilität und Leistungsfähigkeit charakterisiert, wodurch Nutzer die Skalierbarkeit und plattformübergreifenden Fähigkeiten der TensorFlow-Plattform in vollem Umfang nutzen können⁴.

²<https://github.com/tensorflow/tensorflow>

³<https://www.tensorflow.org/guide/keras>

⁴<https://keras.io/about/>

3

Literaturreview

Hier wird der Inhalt der Arbeit präsentiert.

4

Problembeschreibung

4.1. Datenbeschaffung und -vorverarbeitung

4.2. Architekturentwurf

Pix2Pix

CycleGAN

5

Lösungsbeschreibung

5.1. Training- und Testdaten

Das Training und die Evaluierung von Generative Adversarial Networks (GANs) erfordern die klare Definition von Trainings- und Testdatensätzen. Der entscheidende Unterschied zwischen diesen Datensätzen besteht darin, dass das Modell während des Trainings auf den Trainingsdaten optimiert wird, während die Testdaten verwendet werden, um die Leistung und die Generalisierungsfähigkeiten des Modells zu bewerten.

5.1.1. Datenladung für GAN-Training

Für das effektive Training von GANs ist der Zugriff auf qualitativ hochwertige Datensätze von entscheidender Bedeutung. In diesem Kontext bietet TensorFlow eine umfassende Sammlung öffentlich verfügbarer Datensätze. Die verwendeten Datensätze werden von der Quelle <https://efrosgans.eecs.berkeley.edu> heruntergeladen und lokal extrahiert.

```
1 path_to_zip = tf.keras.utils.get_file(fname=f"{dataset_name}.tar.gz",  
2   origin=_URL, extract=True)
```

Code-Auszug 5.1: Laden eines Datensatzes von einer URL

Die Transformation und Vorverarbeitung der Bilddaten erfolgt durch die TensorFlow-Datensatz-API. Diese API bietet eine effiziente Datenpipeline für das Laden und Verarbeiten von Daten, insbesondere für den Einsatz in Machine-Learning-Modellen. In den folgenden Implementierungen wird ein Dataset durch eine Liste von Dateipfaden als Zeichenketten erzeugt.

```
1 train_horses = tf.data.Dataset.list_files (str(PATH / 'trainA/*.jpg'))
```

Code-Auszug 5.2: Erzeugung eines Tensorflow-Dataset aus CycleGAN Implementierung

5.1.2. Vorverarbeitung des Datensatzes

Um die Leistung der GAN-Modelle zu verbessern, werden vor dem Training Variationen in den Trainingsdaten eingeführt. Dieser Prozess, der Datenjittering und Normalisierung umfasst, trägt dazu bei, das Modell robuster zu machen und eine bessere Konvergenz während des Trainings zu erreichen.

//TODO Normalisierung auf $[-1:1]$, begründung

```

1 def normalize(image):
2     image = (image / 127.5) - 1
3     return image

```

Code-Auszug 5.3: Vorverarbeitung des Datensatzes: Normalisierung

Datenjittering wird durchgeführt, indem die heruntergeladenen Bilder (256x256) auf eine größere Größe (286x286) mit der Nearest-Neighbor-Methode skaliert. Bei dieser Methode wird für jedes Pixel im skalierten Bild der Farbwert des nächstgelegenen Pixels im Originalbild übernommen, um eine einfache und effiziente Skalierung zu ermöglichen. Das Bilder werden daraufhin zufällig zugeschnitten. Darüber hinaus wird eine zufällige Spiegelung angewendet.

```

1 def random_crop(image):
2     cropped_image = tf.image.random_crop(image, size=(
3         IMG_HEIGHT, IMG_WIDTH, 3))
4     return cropped_image
5
6 def random_jitter(image):
7     image = tf.image.resize(image, [286, 286], method=tf.image.
8         ResizeMethod.NEAREST_NEIGHBOR)
9     image = random_crop(image)
10    image = tf.image.random_flip_left_right(image)
11    return image

```

Code-Auszug 5.4: Vorverarbeitung des Datensatzes: Jittering

Zusätzlich dazu werden die Bildpfade geladen und in das resultierende JPEG-Format decodiert.

```

1 def load_image(image_path):
2     image = tf.io.read_file(image_path)
3     image = tf.io.decode_jpeg(image, channels=3)
4     image = tf.cast(image, tf.float32)
5     return image

```

Code-Auszug 5.5: Lesen eines Bildes

Die vorverarbeiteten Bilder werden dann in TensorFlow-Datasets integriert. Nachfolgend wird der Trainingsdatensatz zufällig gemischt und in Batches gruppiert, wodurch sichergestellt wird, dass das Modell nicht von der Reihenfolge der Datenpunkte beeinflusst wird.

```
1 def preprocess_image_train(image_path):
2     image = load_image(image_path)
3     image = random_jitter(image)
4     image = normalize(image) if not tf.reduce_all(tf.math.logical_and(
5         image >= 0.0, image <= 1.0)) else image
6     return image
7
8 def preprocess_image_test(image_path):
9     image = load_image(image_path)
10    image = normalize(image) if not tf.reduce_all(tf.math.logical_and(
11        image >= 0.0, image <= 1.0)) else image
12    return image
13
14 train_dataset = tf.data.Dataset.list_files(str(PATH / 'train/*.jpg'))
15 train_dataset = train_dataset.map(preprocess_image_train,
16     num_parallel_calls=tf.data.AUTOTUNE)
17 train_dataset = train_dataset.shuffle(BUFFER_SIZE)
18 train_dataset = train_dataset.batch(BATCH_SIZE)
```

Code-Auszug 5.6: Integration der vorverarbeiteten Bilder in Tensorflow-Datasets

Diese umfassende Vorverarbeitung stellt sicher, dass das GAN-Modell auf optimal vorbereiteten Daten trainiert wird, um eine maximale Leistung und Generalisierungsfähigkeit zu erreichen.

5.2. Implementierung der Pix2PixGAN-Architektur

5.2.1. Generator

Die Struktur des Generator in der Pix2Pix-Implementierung ist ein wesentlicher Aspekt, der die Leistungsfähigkeit des Modells bestimmt. Der Generator ist als U-Net-Architektur aufgebaut, die aus dem Encoder und Decoder bestehen die wiederum aufeinanderfolgende Downsampling- und Upsampling-Schritten beinhalten. Im Encoder-Teil des Generators wird das Downsampling durch eine Reihe von Convolutional Neuronal Network (CNN) Schichten realisiert, die durch die downsample-Funktion innerhalb des downstack definiert sind.

```

1      down_stack = [
2          downsample(64, 4, apply_batchnorm=False), # (batch_size, 128,
              128, 64)
3          downsample(128, 4), # (batch_size, 64, 64, 128)
4          downsample(256, 4), # (batch_size, 32, 32, 256)
5          downsample(512, 4), # (batch_size, 16, 16, 512)
6          downsample(512, 4), # (batch_size, 8, 8, 512)
7          downsample(512, 4), # (batch_size, 4, 4, 512)
8          downsample(512, 4), # (batch_size, 2, 2, 512)
9          downsample(512, 4), # (batch_size, 1, 1, 512)
10     ]

```

Code-Auszug 5.7: Downsampling-Schritt

Die `downsample`-Funktion erstellt eine Downsampling-Schicht, die mittels einer Conv2D-Schicht mit spezifischen Filtern und Kernel-Größen die räumlichen Dimensionen der Eingabebilder reduziert. Zur Verbesserung der Stabilität und Leistungsfähigkeit des Modells integriert die Funktion optional eine Batch-Normalisierung. Diese Normalisierung reguliert und standardisiert die Ausgabe der Conv2D-Schicht, was dazu beiträgt, das Training effizienter zu gestalten. Darüber hinaus beinhaltet die Funktion eine LeakyReLU-Aktivierung, eine Variation der herkömmlichen ReLU-Aktivierungsfunktion. LeakyReLU ermöglicht es, dass auch für negative Eingabewerte ein kleiner Gradient erhalten bleibt, wodurch das Problem der inaktiven Neuronen, bekannt als "sterbende ReLUs", vermieden wird.

```

1      def downsample(filters, size, apply_batchnorm=True):
2          initializer = tf.random_normal_initializer(0., 0.02)
3
4          result = tf.keras.Sequential()
5          result.add(
6              tf.keras.layers.Conv2D(filters, size, strides=2, padding='same',
              ,
7              kernel_initializer=initializer, use_bias=False))
8
9          if apply_batchnorm:
10             result.add(tf.keras.layers.BatchNormalization())
11
12             result.add(tf.keras.layers.LeakyReLU())
13
14             return result

```

Code-Auszug 5.8: Downsampling-Schicht

Im Anschluss daran erfolgt das Upsampling im Decoder-Teil des Generators, das

durch die upsample-Funktion innerhalb des upstack repräsentiert wird. Diese Schichten arbeiten daran, die Merkmale auf ein höher aufgelöstes Format zu projizieren und die Bildgröße wiederherzustellen.

```
1      up_stack = [  
2      upsample(512, 4, apply_dropout=True), # (batch_size, 2, 2,  
        1024)  
3      upsample(512, 4, apply_dropout=True), # (batch_size, 4, 4,  
        1024)  
4      upsample(512, 4, apply_dropout=True), # (batch_size, 8, 8,  
        1024)  
5      upsample(512, 4), # (batch_size, 16, 16, 1024)  
6      upsample(256, 4), # (batch_size, 32, 32, 512)  
7      upsample(128, 4), # (batch_size, 64, 64, 256)  
8      upsample(64, 4), # (batch_size, 128, 128, 128)  
9      ]
```

Code-Auszug 5.9: Upsampling-Schritt

Die upsampling-Funktion verwendet eine spezielle Art von konvolutioneller Schicht, die Conv2DTranspose-Schicht, um die Bildgröße zu erhöhen. Diese Schicht kehrt den Prozess einer konvolutionellen Schicht um, indem sie die Eingabedaten expandiert, was für das Wiederherstellen einer größeren Bildgröße im Generator unerlässlich ist. Zusätzlich zur Conv2DTranspose-Schicht integriert die upsample-Funktion eine Batch-Normalisierung, die zur Stabilisierung des Lernprozesses beiträgt, indem sie die Ausgaben der Conv2DTranspose-Schicht normalisiert. Dies ist ein wichtiger Schritt, um die interne Kovariantenverschiebung zu reduzieren und die Leistung des Modells zu verbessern. Ein weiteres wichtiges Merkmal der Funktion ist die optional Anwendung von Dropout. Wenn diese aktiviert ist, hilft Dropout, Überanpassungen (Overfitting) zu vermeiden, indem zufällig eine bestimmte Anzahl von Neuronen während des Trainingsprozesses ausgeschaltet werden. Dies trägt dazu bei dass das Modell robustere und generalisierbare Merkmale lernt. Schließlich wird eine ReLU-Aktivierungsfunktion angewendet, die dafür sorgt, dass das Modell nicht-lineare Zusammenhänge lernt.

```

1  def upsample(filters, size, apply_dropout=True):
2      initializer = tf.random_normal_initializer(0., 0.02)
3
4      result = tf.keras.Sequential()
5      result.add(
6          tf.keras.layers.Conv2DTranspose(filters, size, strides=2,
7          padding='same',
8          kernel_initializer=initializer,
9          use_bias=False))
10
11     result.add(tf.keras.layers.BatchNormalization())
12
13     if apply_dropout:
14         result.add(tf.keras.layers.Dropout(0.5))
15
16     result.add(tf.keras.layers.ReLU())
17
18     return result

```

Code-Auszug 5.10: Upsampling-Schritt

Die Skip-Verbindungen werden im Generator durch die Speicherung und spätere Verwendung der Ausgaben der Downsampling-Schichten in der skips-Liste realisiert. Nach dem Downsampling-Prozess werden diese gespeicherten Ausgaben in umgekehrter Reihenfolge durchlaufen und mit den Ausgaben der Upsampling-Schichten mittels einer Concatenate-Operation verbunden. Diese Kombination von hoch- und niedrigstufigen Merkmalen führt zu einer detaillierteren und genaueren Bildrekonstruktion.

Schließlich wird das endgültige Bild durch die letzte Schicht des Generators erzeugt, eine Conv2DTranspose-Schicht, die die Ausgabe des Generators darstellt. Diese letzte Schicht spielt eine entscheidende Rolle bei der Erzeugung des endgültigen Bildes, das die kombinierten Merkmale aus dem gesamten Netzwerk nutzt.

```

1      initializer = tf.random_normal_initializer(0., 0.02)
2      last = tf.keras.layers.Conv2DTranspose(
3          OUTPUT_CHANNELS, 4,
4          strides=2,
5          padding='same',
6          kernel_initializer = initializer ,
7          activation='tanh') # (batch_size, 256, 256, 3)
8
9      x = inputs
10
11     # Downsampling through the model
12     skips = []
13     for down in down_stack:
14         x = down(x)
15         skips.append(x)
16
17     skips = reversed(skips[:-1])
18
19     # Upsampling and establishing the skip connections
20     for up, skip in zip(up_stack, skips):
21         x = up(x)
22         x = tf.keras.layers.Concatenate()([x, skip])
23
24     x = last(x)
25
26     return tf.keras.Model(inputs=inputs, outputs=x)

```

Code-Auszug 5.11: Skip Verbindungen

5.2.2. Diskriminator

Der Diskriminator startet mit der Initialisierung seiner Eingabeschichten. Er empfängt zwei separate Bilder - ein Eingabebild (*inp*) und ein Zielbild (*tar*). Diese Bilder werden dann entlang ihrer Farbkanäle zu einem einzigen Bild zusammengefügt.

Im Kern des Diskriminators des Pix2Pix-Modells finden sich mehrere Downsampling- oder Konvolutionsschichten, die eine Schlüsselrolle bei der Bewertung des Eingabebildes spielen. Jede dieser Schichten führt Konvolutionen durch, um die Merkmale und Texturen aus den Bildern zu extrahieren. Dabei arbeitet jede Konvolutionsschicht mit einem kleinen Bereich des Eingabebildes. Dieser Bereich gleitet über das gesamte Bild und bewertet bei jedem Schritt einen kleinen Teil, bekannt als "Patch". Die Größe dieses Bereiches und damit die Größe des bewerteten Patches,

wird durch die Größe des Konvolutionskerns bestimmt. Diese konsequente Analyse von Patches ermöglicht es dem Diskriminator, die räumliche Auflösung des Bildes schrittweise zu reduzieren, was wiederum die Komplexität des Problems verringert und eine effektive Bewertung der lokalen Bildmerkmale ermöglicht.

Nach den Downsampling-Schritten folgen Zero Padding und zusätzliche Konvolutionsschichten. Diese Schritte sind entscheidend, um den Diskriminator zu ermöglichen, feinere Details aus den Bildern herauszuarbeiten.

Die letzte Schicht im Diskriminator ist eine Konvolutionsschicht, die eine Karte von Werten erzeugt. Jeder dieser Werte repräsentiert das Urteil des Diskriminators über einen bestimmten Patch des Bildes.

Der Schlüssel des Diskriminators liegt im PatchGAN-Konzept. Dieses Konzept geschieht implizit durch die Art und Weise, wie die Konvolutionsschichten im Netzwerk strukturiert sind.

5.2.3. Verlustfunktion

Generatorverlust

Der Generatorverlust im Pix2Pix-Modell besteht aus zwei wesentlichen Komponenten: dem adversariellen Verlust (*gan_loss*) und dem L1-Verlust (*l1_loss*). Der adversarielle Verlust wird durch Anwendung der *BinaryCrossentropy*-Funktion von TensorFlow ermittelt. Hierbei wird die Ausgabe des Diskriminators für generierte Bilder (*disc_generated_output*) mit einem Tensor, der ausschließlich aus Einsen besteht, verglichen. Dies dient dazu, die Effektivität des Generators bei der Erzeugung von Bildern zu bewerten, die für den Diskriminator von echten Bildern nicht unterscheidbar sind. Der L1-Verlust hingegen berechnet den mittleren absoluten Fehler zwischen dem vom Generator erzeugten Bild (*gen_output*) und dem tatsächlichen Zielbild (*target*). Dieser Verlust trägt maßgeblich dazu bei, die inhaltliche Übereinstimmung und Ähnlichkeit des generierten Bildes mit dem Zielbild zu fördern.

Der Gesamtverlust des Generators (*total_gen_loss*) ist die Summe des adversariellen Verlust und des L1-Verlusts, wobei der L1-Verlust mit einem Faktor *LAMBDA* gewichtet wird. Die Gewichtung des L1-Verlusts hilft dabei, die strukturelle Integrität und die Genauigkeit des generierten Bildes zu verbessern, indem sie darauf abzielt, die Pixeldifferenzen zwischen dem generierten Bild und dem Zielbild zu minimieren.

Nach der Berechnung des Gesamtverlusts werden Gradienten bezüglich der Generatorparameter berechnet (*gen_tape.gradient*) und diese Gradienten werden dann verwendet, um den Generator mittels des Adam-Optimierers (*generator_optimizer.apply_gradients*) zu aktualisieren. Dieser Prozess ist ein integraler Bestandteil des Trainings, da er dem Generator hilft, sich schrittweise zu verbessern und immer realistischere Bilder zu erzeugen.

Diskriminatorverlust

Der Diskriminatorverlust wird durch die *discriminator_loss*-Funktion im Coe bestimmt. Diese Funktion enthält zwei Eingaben: *disc_real_output*, die Diskriminatorausgabe für das echte Bild und *disc_generated_output*, die Diskriminatorausgabe für das vom Generator erzeugte Bild. Der Verlust für echte Bilder (*real_loss*) wird berechnet, indem die *BinaryCrossentropy*-Funktion zwischen *disc_real_output* und einem Tensor aus Einsen angewendet wird. Dieser Schritt bewertet, wie gut der Diskriminator echte Bilder als solche erkennen kann. Der Verlust für generierte Bilder (*generated_loss*) wird berechnet, indem die *BinaryCrossentropy*-Funktion zwischen *disc_generated_output* und einem Tensor aus Nullen angewendet wird. Dies bewertet, wie gut der Diskriminator generierte Bilder als falsch erkennen kann. Der Gesamtverlust des Diskriminators (*total_disc_loss*) ist die Summe von *real_loss* und *generated_loss*. Diese Kombination zwingt den Diskriminator, zwischen echten und generierten Bildern besser zu unterscheiden.

Für die Optimierung des Diskriminator werden die Gradienten des Diskriminatorverlusts in Bezug auf die Diskriminatorparameter berechnet (*disc_tape.gradient*).

Diese Gradienten werden dann verwendet, um den Diskriminator mittels des Adam-Optimierers (`discriminator_optimizer.apply_gradients`) zu aktualisieren.

5.2.4. Training und Hyperparameter

Der Trainingsprozess des Pix2Pix-Modells ist iterativ und umfasst die abwechselnde Anpassung des Generators und des Diskriminators. Während des Trainings werden Paare von Eingabe- und Zielbildern verwendet, die aus dem Datensatz geladen, vorverarbeitet und dem Modell zugeführt werden. Für jeden Trainingsschritt wird der Generator verwendet, um aus den Eingabebildern Bilder zu erzeugen, die dann vom Diskriminator bewertet werden. Der Diskriminator wird ebenfalls mit den echten Zielbildern trainiert, um die Unterscheidungsfähigkeit zwischen echten und generierten Bildern zu verbessern. Der Trainingsschritt beinhaltet die Berechnung und Anwendung von Gradienten sowohl für den Generator als auch für den Diskriminator, basierend auf ihre jeweiligen Verlustfunktionen.

Für den Generator und den Diskriminator, wird der Adam-Optimierer mit einer Lernrate von 0.0002 und den Momentum-Parametern $\beta_1 = 0.5$ und $\beta_2 = 0.999$ verwendet. Diese Einstellungen einen guten Kompromiss zwischen der Lerngeschwindigkeit und der Stabilität des Trainingsprozesses zu finden. Die Batchgröße ist im Code auf 1 gesetzt. Eine kleine Batchgröße kann zu einer höheren Stabilität im Trainingsprozess beitragen. Der Hyperparameter *LAMBDA* wird verwendet, um das Gewicht des L1-Verlustes im Generatorverlust zu steuern. Ein hoher Wert von *LAMBDA* betont die Bedeutung der Inhaltsähnlichkeit zwischen den generierten und den Zielbildern. Die Anzahl der Trainingsepochen ist auf 450 gesetzt, was darauf hindeutet, dass das Modell eine umfangreiche Trainingsdauer durchläuft, um eine optimale Leistung zu erreichen.

5.3. Implementierung der CycleGAN-Architektur

Nach den theoretischen Grundlagen der CycleGAN-Architektur und der Datenverarbeitung in den vorherigen Abschnitten, wird nun die konkrete Implementierung der Architekturen unter Verwendung von TensorFlow und Keras vorgestellt. Die Generator- und Diskriminatorarchitekturen wurden entsprechend der im Grundlagenkapitel erklärten theoretischen Grundlagen umgesetzt, insbesondere den Empfehlungen von Zhu et al. (2017) (ZPIE) folgend. Der Generator setzt sich aus einem Encoder-Block, sechs Residualblöcken und einem Dekoder-Block zusammen^{2.3}. Der Diskriminator wurde als sequentielles Modell implementiert und umfasst mehrere Convolutional-Schichten, konzipiert als PatchGAN. Nach jeder Convolutional Schicht beim Generator und Diskriminator, außer bei der letzten, folgt dabei eine Instanznormalisierung und ReLU-Aktivierung. Die spezifischen Implementierungsdetails, inklusive der Helferfunktionen, sind im beigefügten Code zu finden.

5.3.1. Verlustfunktion

Während des Trainings werden verschiedene Verlustfunktionen verwendet, um sicherzustellen, dass der Generator qualitativ hochwertige Bilder generiert und dass die Transformationen zwischen den Domänen konsistent sind. Die zentralen Verlustfunktionen, insbesondere die Gesamtverlustfunktionen für den Generator und den Diskriminator, werden im Folgenden erläutert.

Für die Klassifizierung, ob es sich um echte oder generierte Bilder handelt, wird in der Implementierung der *BinaryCrossentropy*-Verlustfunktion aus TensorFlow/Keras verwendet. Dieser berechnet den binären Kreuzentropieverlust zwischen den Zielwerten und den Vorhersagen ¹.

Gesamtverlust des Generators

Der Gesamtverlust des Generators setzt sich aus dem adversariellen Verlust und dem Zykluskonsistenz-Verlust zusammen. Optional kann der Identitätsverlust berücksichtigt werden, was in der Implementierung erfolgte. Die Integration des Identitätsverlusts stellt sicher, dass das Modell die Struktur des Originalbildes beibehält, was zu einem konsistenteren Transformationsprozess führt. Die spezifischen Implementierungen der adversariellen Verlustfunktion des Generators, des Cycle Consistency Loss und des Identitätsverlusts sind im Code-Anhang zu finden.

Gesamtverlust des Diskriminators

Die Gesamtverlustfunktion des Diskriminators setzt sich aus dem adversariellen Verlust für echte und generierte Bilder zusammen. Der Diskriminator wird entsprechend trainiert, echte Bilder als "1" und generierte Beispiele als "0" zu klassifizieren. Die Multiplikation des Gesamtverlustes mit dem Wert 0.5 bildet den Durchschnitt des Gesamtverlustes und stellt sicher, dass die Gradientenaktualisierung während des Prozesses angemessen skaliert wird.

¹https://www.tensorflow.org/api_docs/python/tf/keras/losses/BinaryCrossentropy

5.3.2. Training und Hyperparameter

Das Training wurde über 300 Epochen durchgeführt, wobei die Modelle durch spezifische Optimierer optimiert wurden. In jedem Durchlauf wurde ein Bild aus den Trainingsdaten ausgewählt, und der Generator wurde auf dieses angewendet, um inkrementell die Verbesserung zu verfolgen. Die generierten Bilder wurden mittels *plt.savefig()* aus der Matplotlib-Bibliothek gespeichert, um auch nach dem Training darauf zugreifen zu können.

Diese Vorgehensweise ermöglicht eine systematische Überprüfung der Modellleistung und eine visuelle Darstellung der Fortschritte während des Trainingsprozesses. Die Wahl der Hyperparameter, einschließlich Lernraten und des Lambda-Werts für den Cycle Consistency Loss, wurde durch abgestimmte Experimente ermittelt. Die Optimierer wurden sorgfältig ausgewählt, um eine effiziente Konvergenz während des Trainings zu gewährleisten.

```

1      def Discriminator():
2          initializer = tf.random_normal_initializer(0., 0.02)
3
4          inp = tf.keras.layers.Input(shape=[256, 256, 3], name='
              input_image')
5          tar = tf.keras.layers.Input(shape=[256, 256, 3], name='
              target_image')
6
7          x = tf.keras.layers.concatenate([inp, tar]) # (
              batch_size, 256, 256, channels*2)
8
9          down1 = downsample(64, 4, False)(x) # (batch_size, 128,
              128, 64)
10         down2 = downsample(128, 4)(down1) # (batch_size, 64,
              64, 128)
11         down3 = downsample(256, 4)(down2) # (batch_size, 32,
              32, 256)
12
13         zero_pad1 = tf.keras.layers.ZeroPadding2D()(down3) # (
              batch_size, 34, 34, 256)
14         conv = tf.keras.layers.Conv2D(512, 4, strides=1,
15             kernel_initializer = initializer ,
16             use_bias=False)(zero_pad1) # (batch_size, 31, 31, 512)
17
18         batchnorm1 = tf.keras.layers.BatchNormalization()(conv)
19
20         leaky_relu = tf.keras.layers.LeakyReLU()(batchnorm1)
21
22         zero_pad2 = tf.keras.layers.ZeroPadding2D()(leaky_relu)
23             # (batch_size, 33, 33, 512)
24
25         last = tf.keras.layers.Conv2D(1, 4, strides=1,
26             kernel_initializer = initializer )(zero_pad2) # (
              batch_size, 30, 30, 1)
27
28         return tf.keras.Model(inputs=[inp, tar], outputs=last)

```

Code-Auszug 5.12: Diskriminator

```
1 loss_obj = tf.keras.losses.BinaryCrossentropy(from_logits=True)
```

Code-Auszug 5.13: Initialisierung des
BinaryCrossentropy-Verlustfunktion

```
1 def generator_loss(real_x, cycled_x, real_y, cycled_y, identity,
2   discriminator_generated, step):
3   gan_loss = generator_adversarial_loss(discriminator_generated)
4   cycle_loss = cycle_consistency_loss(real_x, cycled_x) +
5     cycle_consistency_loss(real_y, cycled_y)
6   id_loss = identity_loss(real_x, identity)
7   total_loss = gan_loss + cycle_loss + id_loss
8   return total_loss
```

Code-Auszug 5.14: Vorverarbeitung des Datensatzes: Jittering

```
1 def discriminator_adversarial_loss(real, generated):
2   real_loss = loss_obj(tf.ones_like(real), real)
3   generated_loss = loss_obj(tf.zeros_like(generated), generated)
4   total_disc_loss = real_loss + generated_loss
5   return total_disc_loss * 0.5
```

Code-Auszug 5.15: Vorverarbeitung des Datensatzes: Jittering

```
1 def Generator():
2     inputs = tf.keras.layers.Input(shape=[None,None,3])
3
4     # Layer 1
5     x = layers.Conv2D(64, 7, strides=1, padding='same')(inputs)
6     x = layers.BatchNormalization()(x)
7     x = layers.Activation('relu')(x)
8
9     # Layer 2+ 3
10    x = convolutional_layer(x, 128, 3,2)
11    x = convolutional_layer(x, 256, 3, 2)
12
13    # Residual Blocks
14    for _ in range(6):
15        x = residual_block(x, 256)
16
17    # Layer 10 + 11
18    x = t_convolutional_layer(x, 128, 3, 2)
19    x = t_convolutional_layer(x, 64, 3, 2)
20
21    # Layer 12
22    outputs = layers.Conv2D(3, 7, strides=1, padding='same', activation
        ='tanh')(x)
23
24    return tf.keras.Model(inputs=inputs, outputs=outputs)
```

Code-Auszug 5.16: CycleGAN Generator in Tensorflow

```
1 def Discriminator():
2     model = tf.keras.Sequential()
3
4     # Layer 1
5     model.add(layers.Conv2D(64, 4, strides=2, padding='same',
6         input_shape=(256,256,3)))
7     model.add(layers.BatchNormalization())
8     model.add(layers.Activation('relu'))
9
10    # Layer 2
11    model.add(layers.Conv2D(128, 4, strides=2, padding='same'))
12    model.add(layers.BatchNormalization())
13    model.add(layers.Activation('relu'))
14
15    # Layer 3
16    model.add(layers.Conv2D(256, 4, strides=2, padding='same'))
17    model.add(layers.BatchNormalization())
18    model.add(layers.Activation('relu'))
19
20    # Layer 4
21    model.add(layers.Conv2D(512, 4, strides=2, padding='same'))
22    model.add(layers.BatchNormalization())
23    model.add(layers.Activation('relu'))
24
25    # Layer 5
26    model.add(layers.Conv2D(1, 4, strides=1, padding='same'))
27    model.add(layers.BatchNormalization())
28    model.add(layers.Activation('sigmoid'))
29    return model
```

Code-Auszug 5.17: CycleGAN Diskriminator in Tensorflow

6

Evaluation

6.1. Bewertungskriterien

6.2. Pix2Pix: Ergebnisse und objektive Bewertung

6.3. CycleGAN: Ergebnisse und objektive Bewertung

6.4. Vergleich und Bewertung von Pix2Pix und CycleGAN

6.5. Vergleich von Pix2Pix und CycleGAN

- Matching Paare von Bildern sind ebenfalls für das Training nicht nötig (crewall)
- Macht die Datenvorbereitung einfacher und öffnet neue Techniken für Applikationen (crewall)

7

Fazit und Ausblick

Hier wird ein Fazit und ein Ausblick gegeben.

7.1. Fazit

Fazit.

7.2. Ausblick

Ausblick.



Anhang - Code

Hier sehen Sie den gesamten Quellcode!

B

Anhang - Dokumentationen

Hier sehen Sie die gesamten Dokumentationen zu den erstellten Programmen.

Literaturverzeichnis

- [.]
- [.2010] *Proceedings of the 27th International Conference on International Conference on Machine Learning*. Madison, WI, USA : Omnipress, 2010 (ICML'10). – ISBN 9781605589077
- [.2019] *Proceedings of The 7th International Conference on Intelligent Systems and Image Processing 2019*. The Institute of Industrial Application Engineers, 2019 . – ISBN 9784907220198
- [AMB21] AGGARWAL, Alankrita ; MITTAL, Mamta ; BATTINENI, Gopi: Generative adversarial network: An overview of theory and applications. In: *International Journal of Information Management Data Insights* 1 (2021), Nr. 1, S. 100004. <http://dx.doi.org/10.1016/j.ijjimei.2020.100004>. – DOI 10.1016/j.ijjimei.2020.100004. – ISSN 26670968
- [CWD⁺18] CRESWELL, Antonia ; WHITE, Tom ; DUMOULIN, Vincent ; ARULKUMARAN, Kai ; SENGUPTA, Biswa ; BHARATH, Anil A.: Generative Adversarial Networks: An Overview. In: *IEEE Signal Processing Magazine* 35 (2018), Nr. 1, S. 53–65. <http://dx.doi.org/10.1109/MSP.2017.2765202>. – DOI 10.1109/MSP.2017.2765202. – ISSN 1053–5888
- [Haz21] HAZEM ABDELMOTAAL, AHMED A. ABDOU, AHMED F. OMAR, DALIA MOHAMED EL-SEBAITY, KHALED ABDELAZEEM: Pix2pix Conditional Generative Adversarial Networks for Scheimpflug Camera Color-Coded Corneal Tomography Image Generation. (2021)
- [HB] HUANG, Xun ; BELONGIE, Serge: *Arbitrary Style Transfer in Real-time with Adaptive Instance Normalization*
- [HZRS] HE, Kaiming ; ZHANG, Xiangyu ; REN, Shaoqing ; SUN, Jian: *Deep Residual Learning for Image Recognition*
- [LLW⁺] LIU, Haozhe ; LI, Bing ; WU, Haoqian ; LIANG, Hanbang ; HUANG, Yawen ; LI, Yuexiang ; GHANEM, Bernard ; ZHENG, Yefeng: *Combating Mode Collapse in GANs via Manifold Entropy Estimation*
- [NH10] NAIR, Vinod ; HINTON, Geoffrey E.: Rectified Linear Units Improve Restricted Boltzmann Machines. In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. Madison, WI, USA : Omnipress, 2010 (ICML'10). – ISBN 9781605589077, S. 807–814
- [PJTA] PHILLIP ISOLA ; JUN-YAN ZHU ; TINGHUI ZHOU ; ALEXEI A. EFROS: Image-to-Image Translation with Conditional Adversarial Networks

- [SVR⁺] SRIVASTAVA, Akash ; VALKOV, Lazar ; RUSSELL, Chris ; GUTMANN, Michael U. ; SUTTON, Charles: *VEEGAN: Reducing Mode Collapse in GANs using Implicit Variational Learning*
- [UVL] ULYANOV, Dmitry ; VEDALDI, Andrea ; LEMPITSKY, Victor: *Instance Normalization: The Missing Ingredient for Fast Stylization*
- [ZGQZ19] ZHU, Miao M. ; GONG, Shengrong ; QIAN, Zhenjiang ; ZHANG, Lifeng: A Brief Review on Cycle Generative Adversarial Networks. In: *Proceedings of The 7th International Conference on Intelligent Systems and Image Processing 2019*, The Institute of Industrial Application Engineers, 2019. – ISBN 9784907220198, S. 235–242
- [ZPIE] ZHU, Jun-Yan ; PARK, Taesung ; ISOLA, Phillip ; EFROS, Alexei A.: *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*

Abbildungsverzeichnis

2.1.	Schematische Darstellung der U-Net-Architektur. Die Architektur besteht aus einem Encoder-Teil (links), einem Decoder-Teil (rechts) und Skip-Verbindungen zwischen korrespondierenden Schichten. . .	7
2.2.	Eine mögliche Architektur eines PatchGAN Diskriminator ¹	8
2.3.	Eine Architektur eines CycleGAN Generator. Instanznormalisierung und ReLU Aktivierung erfolgt nach jeder Schicht ¹	11
2.4.	Ein Aufbau eines Residualblocks	12
2.5.	(a) Modell des CycleGANs, bestehend aus zwei Generatoren $F : Y \rightarrow X$ und $G : X \rightarrow Y$ und zugehörige adversarielle Diskriminatoren D_X und D_Y , (b) Cycle-Konsistenz $F(G(x)) \approx x$, (c) Cycle-Konsistenz $G(F(y)) \approx y$ (ZPIE)	13
2.6.	Identity-Mapping für (d) Generator G und (e) Generator F	14

Tabellenverzeichnis

Code-Auszugs-Verzeichnis

5.1. Laden eines Datensatzes von einer URL	21
5.2. Erzeugung eines Tensorflow-Dataset aus CycleGAN Implementierung	21
5.3. Vorverarbeitung des Datensatzes: Normalisierung	23
5.4. Vorverarbeitung des Datensatzes: Jittering	23
5.5. Lesen eines Bildes	23
5.6. Integration der vorverarbeiteten Bilder in Tensorflow-Datasets . . .	24
5.7. Downsampling-Schritt	25
5.8. Downsampling-Schicht	25
5.9. Upsampling-Schritt	26
5.10. Upsampling-Schritt	27
5.11. Skip Verbindungen	28
5.12. Diskriminator	34
5.13. Initialisierung des BinaryCrossentropy-Verlustfunktion	35
5.14. Vorverarbeitung des Datensatzes: Jittering	35
5.15. Vorverarbeitung des Datensatzes: Jittering	35
5.16. CycleGAN Generator in Tensorflow	36
5.17. CycleGAN Diskriminator in Tensorflow	37

Glossar

- **Python:**
Skript- und Programmiersprache, die unter Anderem objektorientiertes Programmieren ermöglicht.
- **Keras:**
Open-Source-API für Deep Learning, seit TensorFlow 2.0 integraler Bestandteil der TensorFlow Core API.
- **TensorFlow:**
Open-Source-Framework für maschinelles Lernen und tiefe neuronale Netzwerke, bekannt für seine Skalierbarkeit und umfangreiche Plattformunterstützung.