eTRainer
Trier University
of Applied Sciences

HOCHSCHULE TRIER

Bachelor-Teamprojekt

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

Analysis, design and implementation of different Generative Adversarial Network 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

Groundbreaking advances in image processing have been achieved through the continuous development of generative adversarial networks (GANs). This thesis deals with the analysis, design and implementation of different GAN architectures. The focus is on the fundamentals of GANs, Pix2Pix and CycleGAN. The emphasis is on the detailed description of the generators and discriminators for Pix2Pix and CycleGAN. This is followed by a comprehensive examination of the implementation and evaluation using various evaluation criteria, including generator and discriminator loss and Structural Similarity Index (SSIM). Finally, a comparison is made between Pix2Pix and CycleGAN to show the respective advantages and disadvantages. The aim of this thesis is to provide a comprehensive overview of the diversity of GAN architectures and identify potential challenges by analysing two representative models.

Zusammenfassung

Bahnbrechende Fortschritte in der Bildverarbeitung wurden durch die kontinuierliche Entwicklung von generativen adversen Netzwerken (GANs) erzielt. Diese Arbeit beschäftigt sich mit der Analyse, dem Entwurf und der Implementierung verschiedener GAN-Architekturen. Der Fokus liegt dabei auf den Grundlagen von GANs, Pix2Pix und CycleGAN. Der Schwerpunkt liegt auf der detaillierten Beschreibung der Generatoren und Diskriminatoren für Pix2Pix und CycleGAN. Es folgt eine umfassende Untersuchung der Implementierung und Evaluierung anhand verschiedener Evaluierungskriterien, darunter Generator- und Diskriminatorverlust und Structural Similarity Index (SSIM). Abschließend wird ein Vergleich zwischen Pix2Pix und CycleGAN durchgeführt, um die jeweiligen Vor- und Nachteile aufzuzeigen. Das Ziel dieser Arbeit ist es, durch die Analyse von zwei repräsentativen Modellen einen umfassenden Überblick über die Vielfalt von GAN-Architekturen zu bieten und mögliche Herausforderungen zu identifizieren.

Abkürzungsverzeichnis

Adam	Adaptive Moment Estimation
CT	Computertomographie
CNN	Convolutional Neural Network
DL	Deep Learning
GAN	Generative Adversarial Network
IS	Interception Score
ML	Machine Learning
MRT	Magnetresonanztomographie
MS	Mode Score
ReLU	Rectified Linear Units
ResNET	Residual Neural Network
SSIM	Structural Similarity Index Measure

Inhaltsverzeichnis

1 Einleitung

In den letzten Jahren haben sich Generative Adversarial Networks (GANs), eine bahnbrechende Entwicklung in der Welt der künstlichen Intelligenz, die maßgeblich auf die Arbeit von Ian Goodfellow und seinem Team zurückgeht, als revolutionäre Methode etabliert. Diese technologische Entwicklung, die erstmals 2014 in einem wegweisenden Paper vorgestellt wurde, hat es ermöglicht, dass Maschinen in der Bildverarbeitung Leistungen erbringen, die früher undenkbar waren. GANs ursprünglich konzipiert als Instrumente zur Erstellung von realistischen Bildern, haben weitreichende Anwendungen in verschiedenen Bereichen gefunden, insbesondere in der Bildverarbeitung, wo sie durch ihre Fähigkeit, Bilder zu generieren und zu transformieren, eine neue Ära eingeleitet haben.
Die innovative Struktur von GANs besteht aus zwei gegeneinander antretenden Netzwerken: dem Generator und dem Diskriminator. Der Generator hat die Aufgabe, Daten zu erzeugen, die von realen Daten ununterscheidbar sind, während der Diskriminator die Echtheit dieser Daten beurteilt. Durch diesen Wettbewerb lernt der Generator, immer bessere Fälschungen zu erzeugen und der Diskriminator wird effizienter im Erkennen dieser Fälschungen. Dieses Zusammenspiel führt zu einer stetigen Verbesserung beider Fälschungen. Dieses Zusammenspiel führt zu einer stetigen Verbesserung beider Netzwerke und ermöglicht die Erstellung hochqualitativer generierter Daten. Diese Methodik bildet den Kern der Funktionsweise von GANs und ist entscheidend für ihre Anwendung in der Bildverarbeitung, wo sie genutzt werden, um realistische Bilder zu generieren und existierende Bilder in vielfältiger Weise zu transformieren (Ian14).
Aufbauend auf dieser grundlegenden Architektur von GANs, hat sich das spezialisierte Framework Pix2Pix als bedeutende Weiterentwicklung in der Welt der Bild-zu-Bild-Übersetzung etabliert. Entwickelt von Phillip Isola und seinem Forschungsteam, adaptiert Pix2Pix das GAN-Prinzip für spezifische Anforderungen der Bildtransformation. Während der klassische Ansatz von GANs auf die Erzeugung neuer, realistisch wirkender Daten abzielt, fokussiert Pix2Pix auf die präzise Übersetzung von Eingangsbildern in gewünschte Ausgangsbilder. Hierbei übernimmt der Generator die Rolle der Bildtransformation, indem er relevante Merkmale aus dem Eingangsbild extrahiert und diese in ein neues, zielgerichtetes Bild umwandelt. Der Diskriminator bewertet anschließend diese Übersetzung, wodurch die Genauigkeit und Realitätsnähe der Bildtransformation weiter optimiert wird. Diese spezialisierte Anwendung von GANs demonstriert die Vielseitigkeit und An-

passungsfähigkeit der Technologie und öffnet neue Horizonte in der Bildverarbeitung (PJTA).
CycleGAN, entwickelt von Jun-Yan Zhu und Kollegen, ist eine Erweiterung der Generative Adversarial Networks (GANs) und speziell für die Aufgabe der Bild-zu-Bild-Übersetzung in Fällen konzipiert, wo keine paarigen Trainingsdaten vorhanden sind. Es unterscheidet sich von herkömmlichen GANs und Pix2Pix durch seinen innovativen Ansatz der SZykluskonsistenz". Im Gegensatz zu Pix2Pix, das paarige Trainingsdaten benötigt (wo jedes Eingangsbild einem spezifischen Ausgangsbild zugeordnet ist), ermöglicht CycleGAN die Übersetzung zwischen zwei Bildsammlungen, ohne dass eine Eins-zu-eins-Beziehung zwischen den Bildern in den Sammlungen besteht. Diese Fähigkeit ist besonders wertvoll für Aufgaben wie das Umwandeln von Sommerbildern in Winterbilder oder das Übertragen von Stilen zwischen verschiedenen Künstlern, wo paarige Trainingsdaten schwierig oder unmöglich zu sammeln sind. Die Schlüsselinnovation von CycleGAN ist die Einführung einer Zykluskonsistenzverlust-Funktion. Diese Funktion sorgt dafür, dass ein Eingangsbild, das in ein Bild einer anderen Domain übersetzt und dann zurück in die ursprüngliche Domain übersetzt wird, dem Originalbild ähnlich bleibt. Zum Beispiel, wenn ein Foto eines Pferdes in ein Zebra umgewandelt wird und dann wieder zurück in ein Pferd, sollte das endgültige Bild dem ursprünglichen Pferdefoto ähnlich sein. Diese Zykluskonsistenz hilft, bedeutungsvolle und kohärente Übersetzungen zwischen unverbundenen Bildsammlungen zu gewährleisten (ZPIE).
In dieser Arbeit wird zunächst ein Überblick über verschiedene GAN-Architekturen gegeben, einschließlich Pix2Pix und CycleGAN, und deren Anwendungen in der Bildverarbeitung untersucht. Darauf folgend wird eine detaillierte Beschreibung der Implementierung verschiedener GAN-Architekturen präsentiert, einschließlich der Behandlung von Trainingsdaten, der Architektur und der Optimierungstechniken.
Der Einsatz von GANs in der Bildverarbeitung bietet vielversprechende Ergebnisse, jedoch unterscheidet sich der Stil und die Herangehensweise dieser Technologie von traditionellen Bildverarbeitungsmethoden. Diese Arbeit zielt darauf ab, die Möglichkeiten von Pix2Pix- und Cycle-GAN in der Bildverarbeitung zu ergründen indem diese Modelle selbst Implementiert und Trainiert werden.
Abschließend werden die Ergebnisse der Durchführung des GAN-Trainings präsentiert und evaluiert. Es wird zudem auch noch untersucht welchen Einfluss die Hyperparameter auf das GAN-Training haben um somit die Resultate des Trainings weiter zu verbessern.
Im Anschluss wird diese Arbeit mit einer Diskussion über die erzielten Ergebnisse und einer Bewertung der verschiedenen GAN-Architekturen abgeschlossen. Ein besonderer Fokus liegt auf der praktischen Anwendung dieser Technologien in der Bildverarbeitung und den Möglichkeiten, die sich daraus für zukünftige Forschungen und Entwicklungen ergeben.

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).

2.1.1. 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).

2.1.2. Training

Das Training besteht in der Optimierung der Parameter für sowohl den Generator als auch den Diskriminator durch die Anwendung von Backpropagation zur Verbesserung dieser Parameter. Dieses Verfahren wird häufig als anspruchsvoll und instabil beschrieben. Einerseits gestaltet sich die Herausforderung, beide Modelle konvergieren zu lassen. Andererseits besteht die Problematik darin, dass der Generator Muster erzeugen kann, die für verschiedene Eingaben äußerst ähnlich sind, was als "Mode-Collapse-Problem"bekannt ist. Der Diskriminatorverlust kann sich rasch gegen Null konvergieren, wodurch ein zuverlässiger Gradientenfluss für die Aktualisierung des Generators verhindert wird. Zur Bewältigung dieser Herausforderungen wurden verschiedene Lösungsansätze vorgeschlagen. Ein Beispiel ist die Verwendung heuristischer Verlustfunktionen. Eine alternative Strategie, die von Sonderby et al. vorgeschlagen wurde, besteht darin, den Datensatz vor der Verwendung zu verrauschen (CWD+18).

Adversarieller Verlust

Der adversarielle Verlust, auch als GAN-Verlust bekannt, spielt eine zentrale Rolle im Trainingsprozess. Dieser Verlust basiert auf dem Konzept des Minimax-Spiels zwischen dem Generator und dem Diskriminator. Der Generator strebt danach, den Diskriminator zu überlisten und Daten zu erzeugen, die von echten Daten nicht zu unterscheiden sind. Gleichzeitig ist es das Ziel des Diskriminators, zwischen echten und generierten Daten zu differenzieren. Der adversarielle Verlust wird in der Gleichung 2.1 repräsentiert.

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{\text{data}}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))] \tag{2.1}$$

Hierbei bezeichnet G den Generator, D den Diskriminator, x echte Daten, z das Rauschen und p_{data} sowie p_z die Wahrscheinlichkeitsverteilungen von echten Daten und Rauschen. Der Minimax-Ansatz impliziert, dass der Generator versucht, den Verlust zu minimieren, während der Diskriminator versucht, ihn zu maximieren. Eine gezielte Optimierung und Anpassung des adversariellen Verlusts ist entscheidend, um Herausforderungen wie dem Mode-Collapse-Problem und den Konvergenzproblemen zu begegnen (HHYY20).

2.1.3. Anwendungen

GANs wurden ursprünglich im Kontext des unüberwachten maschinellen Lernens konzipiert, haben jedoch herausragende Leistungen sowohl im halbüberwachten Lernen als auch im Reinforcement Learning gezeigt (AMB21). Diese Vielseitigkeit hat dazu geführt, dass GANs in verschiedenen Domänen wie dem Gesundheitswesen und dem Bankwesen Anwendung finden.
Im Bereich der medizinischen Bildgebung bieten GANs innovative Lösungsansätze, um den Herausforderungen von Datenknappheit und Patientenprivatsphäre zu begegnen. Von der Erkennung und Behandlung chronischer Krankheiten über die Segmentierung bis hin zur Bildrekonstruktion können GANs vielfältige Anwendungen haben. Zahlreiche GAN-Ansätze wurden bereits entwickelt, um die Rauschunterdrückung in medizinischen Bildverfahren zu verbessern, was wiederum die Qualität von Diagnosen steigern kann (YWB19). Darüber hinaus werden GANs nicht nur im Gesundheitswesen, sondern auch in anderen Bereichen eingesetzt, insbesondere in der Bildsynthese.
In der Finanzindustrie helfen GANs, das Handels- und Risikomanagement zu verbessern, indem sie synthetische Zeitreihen erzeugen, die wichtige Finanzdaten widerspiegeln (EO). Zheng et al. (2018) schlugen beispielsweise eine GAN vor, die auf dem Telekommunikations-Betrugsfall in China im Jahr 2017 basiert und die Wahrscheinlichkeit berechnet, wann eine Überweisung betrügerisch sein könnte (ZZS+18).
Des Weiteren wurden Forschungsanstrengungen unternommen, um mittels GANs menschliche Bewegungen vorherzusagen, insbesondere anhand von 3D-Skelettsequenzen (JZJ+20). Zudem ermöglichen GANs die Identifikation von 3D-Objekten sowie die Generierung realistischer Bilder und Texte in verschiedenen Anwendungsbereichen (AMB21). Die Vielseitigkeit von GANs eröffnet somit ein breites Spektrum an Potenzialen für verschiedene Anwendungsbereiche.

2.1.4. 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

Pix2Pix 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.

2.2.1. Pix2Pix-Kernkonzepte

Architektur des Generators

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 ausgeklügelte 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 fungiert die U-Net-Architektur als zentraler Bestandteil des Generators, verantwortlich für die Bild-zu-Bild-Übersetzung. Die Wahl von U-Net für diesen Zweck basiert auf seiner Fähigkeit, feinere Details und Kontextinformationen aus dem Eingangsbild zu bewahren, was für die Qualität der Bild-zu-Bild-Übersetzung ausschlaggebend ist. Der Generator nutzt die Encoder-Decoder-Struktur von U-Net, um den globalen Kontext des Bildes zu erfassen, während die Skip-Verbindungen sicherstellen, dass auch lokale Details im resultierenden Bild erhalten bleiben. Diese Eigenschaften machen U-Net zu einem effektiven Werkzeug innerhalb des Generators, indem sie eine hohe Detailtreue und Kontextsensitivität in der Bild-zu-Bild-Übersetzung ermöglichen(PJTA).

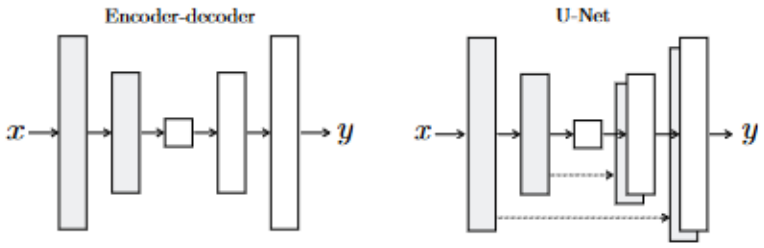


Abbildung 2.1.: Schematische Darstellung der U-Net-Architektur. Die Architektur besteht aus einem Encoder-Teil (links), einem Decoder-Teil (rechts) und Skip-Verbindungen zwischen korrespondierenden Schichten (PJTA).

Architektur des Diskriminator

Im Kontext von Pix2Pix GANs, spielt der PatchGAN-Diskriminator eine besonders wichtige Rolle. Der zentrale Unterschied dieses Diskriminators zu allgemeinen 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 allgemeiner 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 allgemeinen Diskriminatoren auftreten können (PJTA).
Der PatchGAN-Diskriminator kann wenn er effektiv eingesetzt wird, zu besseren und realistischeren Bildern im adverseriellen 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]

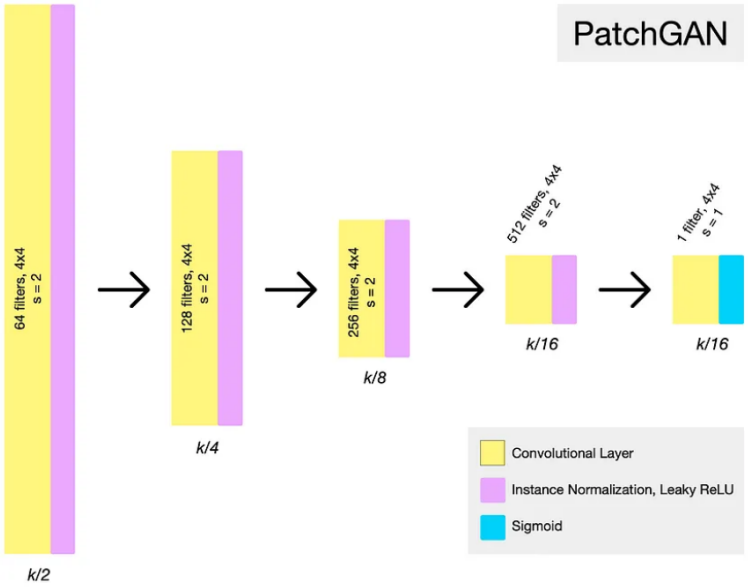


Abbildung 2.2.: Eine mögliche Architektur eines PatchGAN Diskriminator[1]

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

L1-Verlustfunktion

Die L1-Verlustfunktion, auch als Mean Absolute Error (MAE) bekannt, ist im Pix2Pix-Modell, einem bedingten Generative Adversarial Network (cGAN) für Bild-zu-Bild-Übersetzungen, von zentraler Bedeutung. Diese Funktion misst den durchschnittlichen absoluten Unterschied zwischen den vorhergesagten und den tatsächlichen Werten, was besonders bei der Genauigkeit der niedrigen Frequenzen im Bild wichtig ist. Daher trägt die L1-Verlustfunktion maßgeblich zur Bewahrung der strukturellen Integrität und des Kontextes des Bildes bei.
Die Kombination des L1-Verlusts mit dem adversariellen Verlust im Pix2Pix-Modell ist entscheidend. Während der adversarielle Verlust darauf abzielt, die generierten Bilder realistischer wirken zu lassen, fokussiert sich der L1-Verlust auf die niedrigen Frequenzen, um die strukturelle Genauigkeit zu verbessern. Diese Kombination erlaubt es dem Modell, sowohl die niedrigen als auch die hohen Frequenzen effektiv zu erfassen, was zu generierten Bildern führt, die sowohl strukturell korrekt als auch visuell ansprechend sind (PJTA).
Die L1-Verlustfunktion wird allgemein beschrieben als: $L1(G) = \mathbb{E}_{xyz}\left[|y - G(x, z)|\right]$ wobei G das Generator-Modell darstellt, x und z den Eingabewert und den Rauschvektor repräsentieren und y das Ground-Truth-Ergebnis ist. Der L1-Verlust berechnet den durchschnittlichen absoluten Unterschied zwischen den generierten Bildern $G(x, z)$ und den echten Bildern y, was zur Reduzierung der Unschärfe in den generierten Bildern beiträgt (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).
Zusätzlich hat sich die Synergie aus L1- und adversariellen Verlust als vielseitig einsetzbar für diverse Anwendungsfälle im Bereich der Bild-zu-Bild-Übersetzung erwiesen, darunter die semantische Segmentierung und die Farbgebung. Indem es die Pix2Pix-Modellierung ermöglicht, sowohl niedrige als auch hohe Bildfrequenzen präzise zu erfassen, trägt diese Verlustfunktionskombination dazu bei, die visuelle Qualität und die Anwendbarkeit des Modells auf ein breites Spektrum von Herausforderungen zu steigern (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 Verlustfunktion, die speziell auf diese Art der Bildtransformation abgestimmt ist. 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 eins bis drei Kanäle aufweisen, was bedeutet, dass das Modell sowohl mit monochromatischen (Graustufen) als auch mit farbigen Bildern (RGB) arbeiten kann. Im Rahmen des Trainingsprozesses von Pix2Pix wird eine iterative Methode verwendet, bei der der Generator und der Diskriminator abwechselnd traininert werden. Eine Schlüsselkomponente dieses Prozesses ist die Verwendung einer zusammengesetzten Verlustfunktion, die sowohl den adverseriellen 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).
Im Pix2Pix-Modell spielt die Batch-Normalisierung eine wesentliche Rolle für die Stabilisierung des Lernprozesses. Diese Normalisierung ergänzt die iterative Trainingsmethode von Pix2Pix, bei der Generator und Diskriminator abwechselnd trainiert werden. Sie ist besonders wichtig, da der Generator mit einer Vielzahl von Eingabedaten, einschließlich monochromatischer und farbiger Bilder, arbeitet. Zusammen mit der zusammengesetzten Verlustfunktion, die aus adverseriellem und L1-Verlust besteht, verbessert die Batch-Normalisierung die Trainingsstabilität und Qualität der generierten Bilder (PJTA).

2.2.2. Anwendungen von Pix2Pix

Pix2Pix ist eine fortschrittliche Methode für Bild-zu-Bild-Übersetzungsaufgaben und hat eine breite Palette von Anwendungen in der Bildverarbeitung.
Ein markantes Anwendungsbeispiel ist die Umwandlung von architektonische Entwürfe oder Zeichnungen in realistische Gebäudefotos. Besonders eindrucksvoll ist diese Anwendung beim CMP Facades-Datensatz, wo aus simplen Fassadenzeichnungen detailreiche Gebäudebilder generiert werden. (PJTA)
Im Bereich der Fotografie wird Pix2Pix verwendet, um Schwarz-Weiß-Fotos in farbige Bilder zu konvertieren, was besonders bei der Restaurierung alter Fotografien von Bedeutung sein kann (PJTA).
Die Transformation von Tagesaufnahmen in Nachtbilder ist eine weitere beeindruckende Leistung von Pix2Pix sowie die Umwandlung von Thermalaufnahmen in Farbfotos (PJTA).
Schließlich wird Pix2Pix auch zur Vervollständigung von Fotos mit fehlenden Pixeln verwendet, beispielsweise um unvollständige Bilder, die aus Paris StreetView stammen, zu reparieren und zu vervollständigen (PJTA).
Jedoch kann Pix2Pix auch bei medizinischen Anwendungen, insbesondere in der Augenheilkunde, hilfreich sein. Hier wird Pix2Pix für die Post-Interventions-Prognose eingesetzt, um zu zeigen, wie sich die Netzhaut nach einer Anti-VEGF-Injektionsbehandlung verändert. Dies basiert auf Bildern, die vor der Injektion bei Patienten mit exsudativer altersbedingter Makuladegeneration aufgenommen

wurden. Durch diese Technik können Ärzte und Patienten eine Vorstellung davon bekommen, welche Veränderungen im Auge nach der Behandlung zu erwarten sind (Ara22).

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 Zykluskonsistenz-Verlustfunktion (Cycle Consistency Loss), 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 des Generators

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).

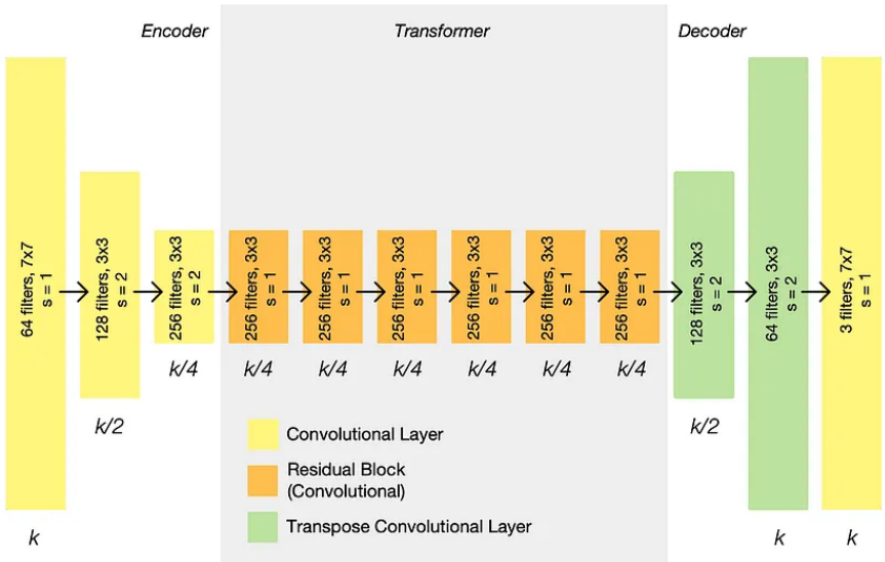


Abbildung 2.3.: Eine Architektur eines CycleGAN Generator. Instanznormalisierung und ReLU Aktivierung erfolgt nach jeder Schicht [1]

[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 einer 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 tiefe neuronale Netze bei Zugabe zusätzlicher Schichten schlechtere Leistungen erbringen als flachere Netze, da die Rückwärtspropagierung von Fehlern in tieferen 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).

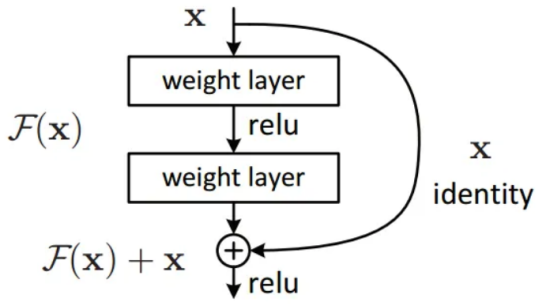


Abbildung 2.4.: Ein Aufbau eines Residualblocks

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

Architektur des Diskriminators

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 $G : X \to Y$ und $F : Y \to X$ 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 2.5. 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. Dies ist in der Abbildung 2.5 dargestellt.
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).

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

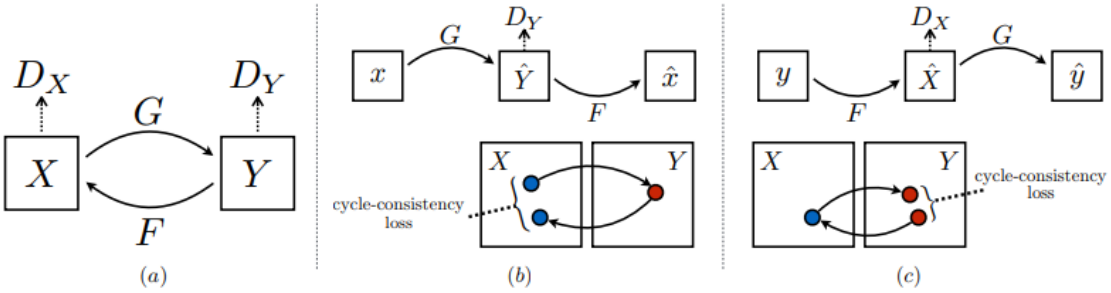


Abbildung 2.5.: (a) Modell des CycleGANs, bestehend aus zwei Generatoren $F : Y \rightarrow X$ und $G : X \rightarrow Y$ und zugehörige adversarielle Diskriminatoren D_X und D_Y, (b) Cycle-Konsistenz $F(G(x)) \approx x$, (c) Cycle-Konsistenz $G(F(y)) \approx y$(ZPIE)

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. Diese Unveränderlichkeit ist visuell veranschaulicht in Abbildung 2.6. 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)

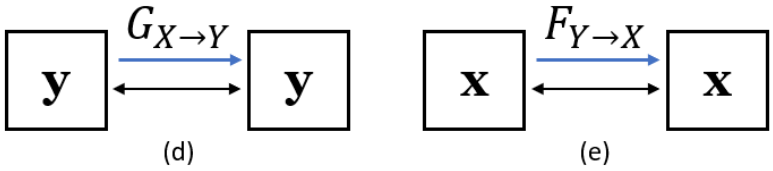


Abbildung 2.6.: Identity-Mapping für (d) Generator G und (e) Generator F

2.3.2. Anwendungen von CycleGAN

CycleGAN hat sich als äußerst vielseitiges Modell erwiesen und findet Anwendung in einer Vielzahl von Bereichen. Insbesondere seine Fähigkeit zur Bildübersetzung zwischen unpaaren Domänen hat zu zahlreichen innovativen Anwendungen geführt. Eine herausragende Nutzung von CycleGAN liegt in der Bild-zu-Bild-Übersetzung und Stilübertragung. Durch dieses Modell können Bilder zwischen verschiedenen Stilen, Szenarien oder Kunstwerken transformiert werden, wodurch die Generierung verschiedener visueller Ästhetiken in einem Bild ermöglicht wird. Diese Anwendung eröffnet kreative Ansätze in der Bildbearbeitung, wie beispielsweise die Umwandlung von Fotografien in den Stil bekannter Kunstwerke (ZPIE). Ein weiterer bedeutender Anwendungsbereich von CycleGAN ist die Gesichtsalterung, die in der Gesichtserkennung mit Altersprogression sowie in forensischen Untersuchungen Anwendung finden kann. Die Fähigkeit des Modells, realistische Altersprogressionen in Gesichtsbildern zu erzeugen, stellt einen innovativen Beitrag zu forensischen Methoden dar (SSJ22).

In der Stenografie eröffnet CycleGAN ebenfalls interessante Anwendungsmöglichkeiten. Hier kann es genutzt werden, um Satellitenbilder in Karten von Google Maps umzuwandeln und umgekehrt. Diese Anwendung zeigt die Anpassungsfähigkeit des Modells im Umgang mit unterschiedlichen Datenmodalitäten (CZS).
Darüber hinaus spielt CycleGAN eine bedeutende Rolle in der medizinischen Bildverarbeitung, indem es die Möglichkeit bietet, Bilder von einer Bildgebungsmodalität in eine andere zu übersetzen, um die Diagnose zu verbessern. Bemerkenswerte Erfolge wurden bereits in der Synthese von MRT-Bildern des Beckenbereichs zu CT-Bildern erzielt (LCS+21).
Die Flexibilität und Vielseitigkeit von CycleGAN machen es zu einem bedeutenden Werkzeug in der Bildverarbeitung, das innovative Möglichkeiten in verschiedenen Branchen eröffnet. Die fortlaufende Erforschung und Anwendung dieses Modells versprechen weiterhin bedeutende Fortschritte in der digitalen Bildtransformation und -interpretation.

2.4. Convolutional Layers

Convolutional Layers repräsentieren eine fundamentale Komponente innerhalb neuronaler Netzwerke, insbesondere im Kontext der Verarbeitung von Bildinformationen. Diese Schicht nutzt Convolution-Operationen, um durch Faltung von Eingabedaten mit Filterkernen lokale Muster zu identifizieren. Die Filterkerne, üblicherweise in Größen wie 3x3, 7x7 oder 9x9, fungieren als kleine Arrays und dienen der Extraktion spezifischer Merkmale im Bild.
Die Funktionsweise dieser Schicht basiert auf der schrittweisen Verschiebung der Filterkerne über das Eingangsbild. An den jeweiligen Pixelpositionen erfolgt eine präzise elementweise Multiplikation, gefolgt von einer anschließenden Summation aller resultierenden Werte. Diese berechneten Werte werden dann in den korrespondierenden Positionen der Feature Map eingetragen, wie in der Abbildung 2.7 veranschaulicht. Durch diesen Prozess gewinnen tiefere Schichten des Netzwerks die Fähigkeit, zunehmend komplexe und abstrakte Informationen auf höheren Ebenen der Hierarchie zu repräsentieren.

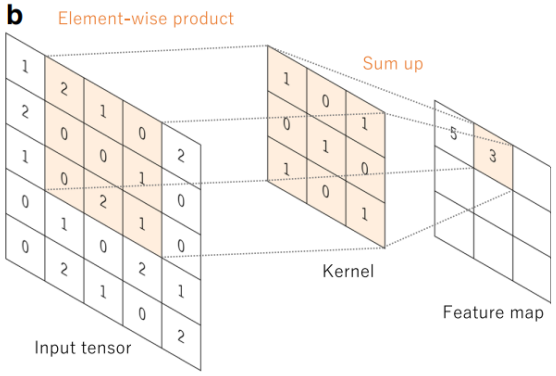


Abbildung 2.7.: Beispiel einer Convolutional-Operation mit einem 3x3 Kernel und Stride 1(YNDT18)

Die Verwendung verschiedener Kernels, sowohl in Bezug auf Größe als auch Anzahl, erlaubt die Extraktion vielfältiger Merkmale wie Kanten oder Texturen. Diese Flexibilität befähigt das Netzwerk, auf unterschiedlichste visuelle Strukturen ansprechend zu reagieren (YNDT18).
Zusätzlich fungieren die Kernels als Subsampling-Mechanismus, bedingt durch die begrenzte Ausdehnung der Convolution-Operation bis zum Bildrand. Die Wahl der Strides, als die Distanz zwischen zwei verschobenen Kernelpositionen, beeinflusst diesen Subsampling-Effekt. Die Anwendung von Padding, um das Bild vor jeder Convolution zu vergrößern, kann dazu beitragen, ungewollte Unterabtastung (Downsampling) zu minimieren.
Ein wesentlicher Vorzug von Convolutional Layers besteht in der signifikanten Reduzierung der zu trainierenden Parameter und der Komplexität des Modells. Dies führt zu einer verbesserten Effizienz, da weniger Gewichte optimiert werden müssen, und trägt zur Prävention von Overfitting bei(YNDT18, ON).
Die Ergebnisse der Convolutional Layer werden anschließend durch eine nichtlineare Aktivierungsfunktion weitergereicht, um die Fähigkeit des Netzwerks zur Modellierung komplexer, nicht-linearer Zusammenhänge zu verbessern. Die Einbeziehung einer Aktivierungsfunktion, wie beispielsweise der ReLU (Rectified Linear Unit), ermöglicht es dem Netzwerk, nicht-linear separierbare Muster und Merkmale zu erfassen. Ohne Aktivierungsfunktionen würden die Convolutional Layers nur lineare Transformationen durchführen, was die Lernfähigkeit des Modells deutlich einschränken würde(SSA17).

2.5. Bibliotheken

2.5.1. Tensorflow

TensorFlow ist ein weit verbreitetes Open-Source-Framework, das für maschinelles Lernen und tiefe neuronale Netzwerke eingesetzt wird[2]. 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 TensorFlow von einer aktiven und engagierten Community, die kontinuierlich zur Weiterentwicklung des Frameworks beiträgt[3].

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 Mo-

[2]https://www.tensorflow.org/
[3]https://github.com/tensorflow/tensorflow

dellierung von Strukturen im Bereich des Deep Learning[4]. Diese Bibliothek, die in der Programmiersprache Python implementiert ist, umfasst sämtliche Phasen des maschinellen Lern-Workflows. Beginnend mit der Datenverarbeitung ermöglicht sie eine Fortführung bis zur präzisen Abstimmung der Hyperparameter während des Trainingsprozesses. Die Keras-Prinzipien zeichnen sich durch Einfachheit, Flexibilität und Leistungsfähigkeit aus und ermöglichen es den Anwendern, die Skalierbarkeit und plattformübergreifenden Fähigkeiten der TensorFlow-Plattform zu nutzen[5].

2.5.3. Matplotlib

Matplotlib ist eine weit verbreitete und leistungsstarke Python-Bibliothek zur Erstellung von statischen, interaktiven und animierten Visualisierungen. Entwickelt von John D. Hunter im Jahr 2003, hat sich Matplotlib zu einem Standardwerkzeug in der Datenvisualisierung und wissenschaftlichen Forschung entwickelt[6].

[4] https://www.tensorflow.org/guide/keras
[5] https://keras.io/about/
[6] https://matplotlib.org/

3 Literaturreview

Die Entstehung und Evolution der Generative Adversarial Networks (GANS) markieren für einen Wendepunkt in der Geschichte der Bildverarbeitung und der künstlichen Intelligenz. Das grundlegende Konzept der GANs wurde 2014 von Ian Goodfellow und seinen Kollegen in ihrem Papier "Generative Adversarial Nets"vorgestellt (Ian14).
In den darauffolgenden Jahren entwickelte sich die GAN-Technologie rasant weiter, weit über die bloße Bildgenerierung hinaus. GANs werden nicht nur für die Generierung von Bildern, sondern auch für die Wiederherstellung von relevanten Informationen in Trainingsdatensätzen sowie für Objekterkennung, Segmentierung und Klassifizierung eingesetzt. Unterschiedliche GAN-Modelle wurden für verschiedene Anwendungsbereiche entwickelt, die jeweils unterschiedliche Stärken aufweisen (AMB21).

Im Rahmen des Papiers 'Image-to-Image Translation with Conditional Adversarial Networks' von Isola et al. (2017) wurde somit das Pix2Pix-Modell vorgestellt, das GANs für die Bild-zu-Bild-Übersetzung nutzt (PJTA). Ebenso erweiterte das von Zhu et al. (2017) präsentierte CycleGAN-Modell die Möglichkeiten der Bild-zu-Bild-Übersetzung für unpaarige Datensätze, was die Flexibilität von GANs in der Anwendung weiter erhöhte (ZPIE). Weitere bedeutende Modelle, die in der Forschung Beachtung finden, sind unter anderem das Conditional GAN (CGAN) (MO), Wasserstein GAN (WGAN) (ACB17) und Deep Convolutional GAN (DCGAN) (KSH17).

Trotz dieser Fortschritte bleibt das Training von GANs eine anspruchsvolle Aufgabe, die verschiedene Herausforderungen mit sich bringt. Um diesen zu begegnen, wurden verschiedene Methoden, Techniken und Architekturen vorgeschlagen. Schwierigkeiten wie instabiles Training und Mode-Kollaps, bei dem Modelle dazu tendieren, eingeschränkte Vielfalt zu erzeugen, werden in der Arbeit von Salimans et al. (2016) (SGZ+) zu Ïmproved Techniques for Training GANsädressiert, um die Stabilisierung der Netzwerke und die Optimierung des Trainingsprozesses zu unterstützen. Eine weiterführende Studie (HHYY20) präsentiert diverse GAN-Architekturen und Methoden, um das Problem des Mode-Kollapses zu überwinden. Dabei werden die Architekturen und Methoden variiert, um die Einsatzmöglichkeiten zu erweitern (JZJ+20, EO, SSJ22) und neue Aspekte der GAN-Technologie

zu erforschen (SVR+).

Im Kontext potenzieller gesellschaftlicher Auswirkungen können GANs erhebliche Herausforderungen im Bereich der Cybersicherheit darstellen, insbesondere hinsichtlich Gesichtserkennung und Passwortentschlüsselung (HGAPC). Aggarwal deutete in seiner Arbeit bereits an, dass GANs in Bezug auf die menschliche Bildsynthese Besorgnis erregen könnten (AMB21). Diese Bedenken eröffnen neue Forschungsrichtungen für GANs, die darauf abzielen, ethische Standards in der Künstlichen Intelligenz zu wahren.
Die fortlaufende Forschung in diesem Bereich strebt danach, die Leistungsfähigkeit von GANs zu verbessern und neue Perspektiven für deren Nutzung in der Bildverarbeitung und künstliche Intelligenz zu erschließen.

4
Problembeschreibung

Generative Adversarial Networks (GANs) haben in den letzten Jahren erhebliche Aufmerksamkeit in der Forschung und Industrie erlangt. Diese neuartige Klasse von künstlichen neuronalen Netzwerken hat das Potenzial, realistische Daten zu generieren und komplexe Probleme in verschiedenen Domänen zu lösen. Im Rahmen dieser Arbeit liegt der Fokus auf zwei spezifischen GAN-Varianten: Pix2Pix und CycleGAN.
Pix2Pix konzentriert sich auf die direkte Zuordnung zwischen Eingabe- und Ausgabebildern, während CycleGAN die Fähigkeit besitzt, nicht paarweise zugeordnete Datensätze zu übersetzen. Diese Modelle sind auf die Generierung von Bildern ausgerichtet und haben das Potenzial, in verschiedenen Szenarien wie der Stilübertragung, der Bildsegmentierung und der Domänenanpassung verwendet zu werden. Trotz ihrer vielversprechenden Anwendungen gibt es jedoch verschiedene Herausforderungen im Design, in der Implementierung und in der Analyse dieser Modelle. Es stellt sich die Frage, wie die Modelle effektiv gestaltet werden können, um optimale Leistung zu erzielen und welche Strategie am besten geeignet sind, um die Modelle erfolgreich zu trainieren und zu evaluieren.

Herausforderungen im Design

Die Entwicklung von GAN, insbesondere von Modellen wie Pix2Pix und CycleGAN, ist mit zahlreichen Herausforderungen verbunden. Die Wahl der Architektur, die Anpassung der Hyperparameter und die Integration von Regularisierungstechniken sind entscheidende Aspekte, die mit besonderer Sorgfalt angegangen werden müssen. Diese Herausforderungen haben einen großen Einfluss auf die Fähigkeit der Modelle, realistische und generalisierte Ergebnisse zu liefern.
Die Wahl der Architektur spielt eine zentrale Rolle und wirkt sich direkt auf die Fähigkeit des Modells aus, komplexe Transformationen und Generierungsaufgaben durchzuführen. Die Anpassung der Hyperparameter erfordert eine Feinabstimmung, um die Konvergenz des Modells ohne Überanpassung zu gewährleisten. Die Integration von Regularisierungstechniken ist von wesentlicher Bedeutung, um das Modell vor Überanpassung zu schützen und seine Gesamtleistung zu verbessern (SC21).
Die Auswirkungen dieser Entscheidungen auf die Leistung und Konvergenz der Modelle sind daher nicht unbedeutend und erfordern eine eingehende Analyse, um

sicherzustellen, dass die GANs in der Lage sind, qualitativ hochwertige und realistische Ergebnisse zu liefern und gleichzeitig eine stabile Konvergenz während des Trainings zu gewährleisten.

Herausforderungen in der Implementierung

Die Auswahl geeigneter Datensätze, der Umgang mit Datenungleichgewichten, die Optimierung der Trainingsparameter und die Vermeidung von Overfitting sind wichtige Schritte bei der Implementierung von Pix2Pix und CycleGAN. Dies umfasst die Datenaufbereitung, das Training und die Evaluierung der Modelle.
Die Auswahl des Datensatzes hat einen großen Einfluss auf die Fähigkeit des Modells, realistische Ergebnisse zu liefern. Dabei ist nicht nur die Menge, sondern auch die Vielfalt der Daten von Bedeutung. Auch der Umgang mit Ungleichgewichten in den Daten ist von hoher Relevanz, um sicherzustellen, dass das Modell nicht in Richtung bestimmter Merkmale verzerrt wird (YNDT18).
Die Optimierung der Trainingsparameter, einschließlich der Lernraten und der Batchgrößen, ist ein Feinabstimmungsprozess, um eine stabile Konvergenz des Modells zu gewährleisten (SC21). Gleichzeitig ist es von entscheidender Notwendigkeit, Overfitting durch die Implementierung geeigneter Regularisierungstechniken zu vermeiden (YNDT18).
Das Verständnis und die zielgerichtete Bewältigung dieser Implementierungsherausforderungen sind entscheidend, um sicherzustellen, dass Pix2Pix und CycleGAN effektiv in verschiedenen Anwendungsbereichen eingesetzt werden können. Durch eine gründliche Untersuchung dieser Aspekte können Modelle entwickelt werden, die nicht nur leistungsfähig, sondern auch robust und generalisierbar sind.

Herausforderungen in der Analyse

Die Analyse von Pix2Pix und CycleGAN umfasst mehrere Schlüsselaspekte, die zum Verständnis der Leistung und Zuverlässigkeit dieser GANs beitragen. Dazu gehören die Bewertung der Generierungsfähigkeiten, die Quantifizierung von Artefakten und die Untersuchung von Knovergenzproblemen.
Die Evaluierung umfasst qualitative Bewertungen der von den Modellen erzeugten Bilder. Dazu können visuelle Inspektionen und Vergleiche mit den Originaldaten durchgeführt werden. Unerwünschte Muster und Unvollkommenheiten können ebenfalls visuell oder mit Hilfe von Metriken identifiziert werden. Dazu gehören Messungen wie der Mode Score (MS), SSIM-Score oder Interception Score (IS) (PYY+19).
Die Analyse kann ferner die Überwachung von Verlustkurven umfassen, um Konvergenzprobleme während des Trainings zu untersuchen.
Analyse und Bewertung sind entscheidend, um die Modelle weiter zu verbessern, ihre Anwendbarkeit zu erweitern und sicherzustellen, dass sie ihren Zweck erfüllen.

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 werden die Datensätze 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 der CycleGAN Implementierung

5.1.2. Vorverarbeitung des Datensatzes

Um die Leistung von GAN-Modellen zu optimieren, werden vor dem Training Variationen in den Trainingsdaten eingeführt. Dieser Prozess umfasst Datenjittering und Normalisierung. Durch die Integration von Variationen wird das Modell robuster, da es eine erhöhte Invarianz gegenüber unterschiedlichen Eingabedaten entwickelt. Dies trägt wesentlich dazu bei, eine verbesserte Konvergenz während des Trainings zu erreichen.

Bei Pix2Pix bestehen die Trainingsdaten aus einem Paar von Eingabe- und Zielbildern, während bei CycleGAN unpaare Daten berücksichtigt werden. Um eine konsistente Skalierung mit der Tanh-Aktivierungsfunktion sicherzustellen, werden diese Bilder im Bereich von -1 bis +1 skaliert. Diese Normalisierung ist von entscheidender Bedeutung für die Stabilisierung des Trainingsprozesses. Durch die Bereitstellung eines standardisierten Datensatzes kann das Modell effektiver mit einer verbesserten Lernrate und Konvergenzgeschwindigkeit arbeiten (RMC).

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

Code-Auszug 5.3: Vorverarbeitung des Datensatzes: Normalisierung

Datenjittering bezieht sich auf die Einführung von zufälligen Variationen oder Veränderungen in den Trainingsdaten, was in der Implementierung durch zufälliges Zuschneiden und eine zufällige Spiegelung erreicht wird. Die heruntergeladenen Bilder mit einer Auflösung von 256x256 werden zuerst auf eine größere Größe von 286x286 skaliert, wobei die Nearest-Neighbor-Methode verwendet wird.

```
1 def random_jitter(image):
2     image = tf.image.resize(image, [286, 286], method=tf.image.
          ResizeMethod.NEAREST_NEIGHBOR)
3     image = tf.image.random_crop(image, size=(286, 286, 3))
4     image = tf.image.random_flip_left_right(image)
5     return image
```

Code-Auszug 5.4: Vorverarbeitung des Datensatzes: Jittering

Diese Methode skaliert Bilder einfach und effizient, indem sie für jedes Pixel im skalierten Bild den Farbwert des nächstgelegenen Pixels im Originalbild übernimmt[1]. Nach dem Skalieren wird das Bild zufällig auf die Originalgröße reduziert und gleichzeitig mit einer Wahrscheinlichkeit von 50% gespiegelt. Anschließend werden

[1] https://en.wikipedia.org/wiki/Nearest-neighbor_interpolation

die Bildpfade mittels der *load_image*-Funktion geladen und in das resultierende JPEG-Format decodiert (Code A.1).

Die vorverarbeiteten Trainings- und Testbilder werden darauf 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(
           image >= 0.0, image <= 1.0)) else image
5      return image
6
7  train_dataset = tf.data.Dataset. list_files (str(PATH / 'trainA/*.jpg'))
8  train_dataset = train_dataset.map(preprocess_image_train,
       num_parallel_calls=tf.data.AUTOTUNE)
9  train_dataset = train_dataset.shuffle(BUFFER_SIZE)
10 train_dataset = train_dataset.batch(BATCH_SIZE)
```

Code-Auszug 5.5: Integration der vorverarbeiteten Trainingsbilder in Tensorflow-Datasets (CycleGAN Implementierung)

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.

```
down_stack = [
        downsample(64, 4, apply_batchnorm=False), # (batch_size, 128,
            128, 64)
        downsample(128, 4), # (batch_size, 64, 64, 128)
        downsample(256, 4), # (batch_size, 32, 32, 256)
        downsample(512, 4), # (batch_size, 16, 16, 512)
        downsample(512, 4), # (batch_size, 8, 8, 512)
        downsample(512, 4), # (batch_size, 4, 4, 512)
        downsample(512, 4), # (batch_size, 2, 2, 512)
        downsample(512, 4), # (batch_size, 1, 1, 512)
]
```

Code-Auszug 5.6: Downsampling-Schritt in Pix2Pix

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 ßterbende ReLUs", vermieden wird.

```
def downsample(filters, size, apply_batchnorm=True):
        initializer = tf.random_normal_initializer(0., 0.02)
        result = tf.keras.Sequential()
        result.add(
        tf.keras.layers.Conv2D(filters, size, strides=2, padding='same'
            , kernel_initializer=initializer, use_bias=False))

        if apply_batchnorm:
                result.add(tf.keras.layers.BatchNormalization())

        result.add(tf.keras.layers.LeakyReLU())
        return result
```

Code-Auszug 5.7: Downsampling-Schicht in Pix2Pix

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 ausgelöstes Format zu projizieren und die Bildgröße wiederherzustellen.

```
up_stack = [
        upsample(512, 4, apply_dropout=True), # (batch_size, 2, 2,
            1024)
        upsample(512, 4, apply_dropout=True), # (batch_size, 4, 4,
            1024)
        upsample(512, 4, apply_dropout=True), # (batch_size, 8, 8,
            1024)
        upsample(512, 4),   # (batch_size, 16, 16, 1024)
        upsample(256, 4),   # (batch_size, 32, 32, 512)
        upsample(128, 4),   # (batch_size, 64, 64, 256)
        upsample(64, 4),   # (batch_size, 128, 128, 128)
]
```

Code-Auszug 5.8: Upsampling-Schritt in Pix2Pix

Die upsampling-Funktion verwendet eine spezielle Art von Convolutional Layer, die Conv2DTranspose-Schicht, um die Bildgröße zu erhöhen. Diese Schicht kehrt den Prozess einer Convolutional Layer Schicht um, indem sie die Eingabedaten expandiert, was für das Wiederherstellen einer größerem 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.

```
def upsample(filters, size, apply_dropout=True):
        initializer = tf.random_normal_initializer(0., 0.02)
        result = tf.keras.Sequential()
        result.add(
        tf.keras.layers.Conv2DTranspose(filters, size, strides=2,
            padding='same', kernel_initializer=initializer,
        use_bias=False))
        result.add(tf.keras.layers.BatchNormalization())

        if apply_dropout:
                result.add(tf.keras.layers.Dropout(0.5))

        result.add(tf.keras.layers.ReLU())
        return result
```

Code-Auszug 5.9: Upsampling-Schritt in Pix2Pix

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 um Overfitting zu vermeiden. 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.

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

Code-Auszug 5.10: Skip Verbindungen in Pix2Pix

5.2.2. Diskriminator

Der Diskriminator startet mit der Initialisierung seiner Eingabeschichten. Er empfängt zwei seperate 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 Convolutional Layer, 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 jeder Convolutional Layer 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 Convolutional Layer. Diese Schritte sind entscheidend, um den Diskriminator zu ermöglichen, feinere Details aus den Bildern herauszuarbeiten.
Die letzte Schicht im Diskriminator ist ein Convolutional Layer, 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 Convolutional Layer im Netzwerk strukturiert sind.

```
def Discriminator():
    initializer = tf.random_normal_initializer(0., 0.02)

    inp = tf.keras.layers.Input(shape=[256, 256, 3], name='input_image')
    tar = tf.keras.layers.Input(shape=[256, 256, 3], name='target_image')

    x = tf.keras.layers.concatenate([inp, tar])  # (batch_size, 256, 256, channels*2)

    down1 = downsample(64, 4, False)(x) # (batch_size, 128, 128, 64)
    down2 = downsample(128, 4)(down1) # (batch_size, 64, 64, 128)
    down3 = downsample(256, 4)(down2) # (batch_size, 32, 32, 256)

    zero_pad1 = tf.keras.layers.ZeroPadding2D()(down3) # (batch_size, 34, 34, 256)
    conv = tf.keras.layers.Conv2D(512, 4, strides=1,
     kernel_initializer=initializer, use_bias=False)(zero_pad1) # (batch_size, 31, 31, 512)

    batchnorm1 = tf.keras.layers.BatchNormalization()(conv)

    leaky_relu = tf.keras.layers.LeakyReLU()(batchnorm1)

    zero_pad2 = tf.keras.layers.ZeroPadding2D()(leaky_relu) # (batch_size, 33, 33, 512)

    last = tf.keras.layers.Conv2D(1, 4, strides=1,
     kernel_initializer=initializer)(zero_pad2) # (batch_size, 30, 30, 1)

    return tf.keras.Model(inputs=[inp, tar], outputs=last)
```

Code-Auszug 5.11: Pix2Pix Diskriminator in Tensorflow

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 eichte 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 flasch 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 Diskriminator-

verlusts 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.

5.3. Implementierung der CycleGAN-Architektur

Nach den theoretischen Grundlagen der CycleGAN-Architektur und der Datenvorverarbeitung in den vorherigen Abschnitten, wird nun die konkrete Implementierung der Architekturen unter Verwendung von TensorFlow und Keras vorgestellt.

5.3.1. Generator und Diskriminator

Die Architekturen des Generators und Diskriminators wurden gemäß den theoretischen Grundlagen umgesetzt, insbesondere den Richtlinien von Zhu et al. (2017) (ZPIE).
Der Generator besteht aus einem Encoder-Block, gefolgt von sechs Residual-Blöcken und einem Decoder-Block, welche in Abbildung 2.3 dargestellt ist. Der Diskriminator wurde als sequentielles Modell implementiert und umfasst mehrere Convolutional Schichten, konzipiert als PatchGAN.
Nach jeder Convolutional Schicht im Generator und Diskriminator, abgesehen von der letzten, wird eine Instanznormalisierung und eine ReLU-Aktivierung angewendet. Die Instanznormalisierung dient dazu, die Aktivierungen zu normalisieren und das Training zu stabilisieren, indem sie die Eingaben jedes Minibatches normalisiert. Die ReLU-Aktivierung fördert die Einführung von Nichtlinearitäten in das Modell und ermöglicht es, komplexere Merkmale zu erfassen.
Für die Output-Schicht wird die tanh-Aktivierungsfuntkion angewendet. Diese Schicht begrenzt die Ausgabewerte auf den Bereich zwischen -1 und +1. Diese Begrenzung ist nicht nur wichtig, um eine konsistente Skalierung mit den Trainingsdaten sicherzustellen, sondern trägt auch zur Stabilisierung des Trainingsprozesses bei (RMC).
Die spezifischen Implementierungsdetails, inklusive der Helferfunktionen, sind im beigefügten Code zu finden.

```python
def Generator():
    inputs = tf.keras.layers.Input(shape=[None,None,3])

    # Layer 1
    x = layers.Conv2D(64, 7, strides=1, padding='same')(inputs)
    x = layers.BatchNormalization()(x)
    x = layers.Activation('relu')(x)

    # Layer 2+ 3
    x = convolutional_layer(x, 128, 3,2)
    x = convolutional_layer(x, 256, 3, 2)

    # Residual Blocks
    for _ in range(6):
        x = residual_block(x, 256)

    # Layer 10 + 11
    x = t_convolutional_layer(x, 128, 3, 2)
    x = t_convolutional_layer(x, 64, 3, 2)

    # Dropout
    x = tf.keras.layers.Dropout(0.5)(x)

    # Layer 12
    outputs = layers.Conv2D(3, 7, strides=1, padding='same',
        activation='tanh')(x)

    return tf.keras.Model(inputs=inputs, outputs=outputs)
```

Code-Auszug 5.12: CycleGAN Generator in Tensorflow

```
def Discriminator():
    model = tf.keras.Sequential()

    # Layer 1
    model.add(layers.Conv2D(64, 4, strides=2, padding='same',
        input_shape=(256,256,3)))
    model.add(layers.BatchNormalization())
    model.add(layers.Activation('relu'))

    # Layer 2
    model.add(layers.Conv2D(128, 4, strides=2, padding='same'))
    model.add(layers.BatchNormalization())
    model.add(layers.Activation('relu'))

    # Layer 3
    model.add(layers.Conv2D(256, 4, strides=2, padding='same'))
    model.add(layers.BatchNormalization())
    model.add(layers.Activation('relu'))

    # Layer 4
    model.add(layers.Conv2D(512, 4, strides=2, padding='same'))
    model.add(layers.BatchNormalization())
    model.add(layers.Activation('relu'))

    # Layer 5
    model.add(layers.Conv2D(1, 4, strides=1, padding='same'))
    model.add(layers.BatchNormalization())
    model.add(layers.Activation('sigmoid'))

    # Dropout
    model.add(tf.keras.layers.Dropout(0.5))
    return model
```

Code-Auszug 5.13: CycleGAN Diskriminator in Tensorflow

5.3.2. 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[2].

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

Code-Auszug 5.14: Initialisierung des BinaryCrossentropy-Verlustfunktion

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 zu einem konsistenteren Transformationsprozess führt (siehe Implementierung 5.15). Die Integration des Identitätsverlusts gewährleistet die Bewahrung der Struktur des Originalbildes.
Im adversariellen Verlust, implementiert durch die Funktion *generator_adversarial_loss*, werden die generierten Bilder mit einem Tensor aus Einsen verglichen, welcher die Zielwerte für 'echt' repräsentiert. Die Verlustberechnung erfolgt mittels einer spezifizierten Verlustfunktion.
Für den Zykluskonsistenz-Verlust (*cycle_loss*-Funktion) wird der mittlere absolute Unterschied zwischen einem echten Bild und seiner zyklisch transformierten Version berechnet. Dieser L1-Verlust stellt sicher, dass die Übersetzung zwischen den Domänen und zurück nahe an der Identitätsabbildung liegt.
Die Funktion *identity_loss* ermittelt den Identitätsverlust, wobei der L1-Verlust zwischen einem echten Bild und seiner übersetzten Version in derselben Domäne verwendet wird.
Zur Berücksichtigung ihrer Bedeutung für den Gesamtverlust wird der Verlust an Zykluskonsistenz und Identität jeweils mit einem gewichteten Faktor multipliziert. Der Code-Anhang enthält detaillierte Implementierungen der genannten Verlustfunktionen.

[2] https://www.tensorflow.org/api_docs/python/tf/keras/losses/BinaryCrossentropy

```
1 def generator_loss(real_x, real_y, cycled_x, cycled_y, disc_fake,
      identity):
2     gen_loss = generator_adversarial_loss(disc_fake)
3     total_cycle_loss = cycle_loss(real_x, cycled_x) + cycle_loss(real_y,
          cycled_y)
4     id_loss = identity_loss(real_y, identity)
5     total_gen_loss = gen_loss + total_cycle_loss + id_loss
6     return total_gen_loss
```

Code-Auszug 5.15: Gesamtverlust des Generators in CycleGAN

Gesamtverlust des Diskriminators

Die Gesamtverlustfunktion des Diskriminators setzt sich aus dem adversariellen Verlust für echte und generierte Bilder zusammen. Der Diskriminator wird dementsprechend trainiert, echte Bilder als Einsen und generierte Beispiele als Nullen zu klassifizieren.
Zu Beginn wird der Verlust berechnet, wenn der Diskriminator echte Bilder betrachtet. Hierbei kommt die BinaryCrossentropy-Verlustfunktion zum Einsatz, die den Verlust zwischen den echten Vorhersagen (*real*) und den Zielwerten berechnet. Anschließend erfolgt die Berechnung des Verlusts zwischen den generierten Vorhersagen (*generated*) und den Zielwerten.
Der Gesamtverlust ergibt sich als Summe der beiden Teilverluste. Um sicherzustellen, dass die Gradientenaktualisierung während des Trainings angemessen skaliert wird, erfolgt eine Multiplikation des Gesamtverlustes mit dem Wert 0.5, was einer Bildung des Durchschnitts des Gesamtverlustes entspricht.

```
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 5.16: Gesamtverlust des Diskriminators in CycleGAN

5.4. Training und Hyperparameter

Während des Trainings von Generative Adversarial Networks (GANs) wird eine iterative Methode angewendet, bei der der Generator und der Diskriminator abwechselnd trainiert werden. Dieser Trainingszyklus erstreckt sich über mehrere Epochen, wobei in jeder Epoche Bilder durch einen Feed-Forward-Prozess vom Generator generiert und vom Diskriminator bewertet werden. Gleichzeitig wird der Diskriminator mit den echten Zielbildern trainiert, um seine Fähigkeit zu verbessern, zwischen echten und generierten Bildern zu unterscheiden. Der Trainingsschritt beinhaltet die Berechnung und Anwendung von Gradienten für sowohl den Generator als auch den Diskriminator, basierend auf ihren jeweiligen Verlustfunktionen (Code A.2, A.9, A.10).

Die Wahl der Hyperparameter, einschließlich der Anzahl der Epochen, Lernraten der Optimizer und der *LAMBDA*-Werte der jeweiligen Architekturen, wurde durch abgestimmte Experimente ermittelt. Dabei wurde darauf geachtet, dass die gewählten Werte zu einer stabilen und konvergenten Schulung führen.

Um Überanpassung entgegenzuwirken, wird auf das Training mit umfangreichen Datensätzen zurückgegriffen. Die Integration zusätzlicher Regularisierung in Form von Dropout trägt dazu bei, die Generalisierungsfähigkeit des Modells zu steigern und seine Robustheit zu verbessern. Beim Dropout-Mechanismus werden während des Trainingsprozesses zufällig bestimmte Neuronen deaktiviert, was dazu führt, dass das Netzwerk nicht übermäßig abhängig von spezifischen neuronalen Pfaden wird. Diese Strategie fördert die Robustheit des Modells, indem sie es dazu zwingt, mit dem Fehlen bestimmter Neuronen umzugehen und alternative Pfade zu nutzen. Durch die erzwungene Redundanz werden verschiedene Teile des Netzwerks aktiviert, was zu einer breiteren Erfassung von Merkmalen und einer vielfältigeren Repräsentation führt. Dies ist entscheidend, um sicherzustellen, dass das Modell nicht zu stark auf spezifische Trainingsdaten reagiert, sondern stattdessen Merkmale erlernt, die auf einer breiteren Palette von Eingaben generalisiert werden können (YNDT18).

5.4.1. Optimierungstechnik und Optimizers

Als Optimierungstechnik wird das Gradientenabstiegsverfahren angewendet, bei dem die Parameter in Richtung des negativen Gradienten der Verlustfunktion aktualisiert werden, um den Verlust zu minimieren. Die Parameter beider Modelle werden mittels korrespondierenden Optimizern aktualisiert. Die Lernrate der Optimizer bestimmt die Geschwindigkeit, mit der das Modell lernt, und sie wird sorgfältig abgestimmt, um eine stabile und konvergente Schulung sicherzustellen.
In der Implementierung wird der Adam-Optimizer verwendet, eine gängige Wahl in der GAN-Literatur. Dieser Optimizer ist eine verbesserte Variante des stochastischen Gradientenabstiegsverfahrens und passt die Lernraten für jede Variable

dynamisch an. Dadurch wird die Konvergenz des Trainingsprozesses beschleunigt und das Modell wird robuster gegenüber unterschiedlichen Lernraten (KB).

```
1 generator_optimizer = tf.keras.optimizers.Adam(learning_rate=0.0002,
      beta_1=0.5)
2 discriminator_optimizer= tf.keras.optimizers.Adam(learning_rate
      =0.0002, beta_1=0.5)
```

Code-Auszug 5.17: Initialisierung der Adam-Optimizers aus der Pix2Pix Implementierung

5.4.2. Fortschrittsüberwachung und Visualisierung

Um inkrementelle Verbesserungen nach jeder Iteration zu visualisieren, wird während des Trainings ein Bild aus dem Trainingsdatensatz ausgewählt, auf das der Generator angewendet wird. Die Eingabebilder und die generierten Bilder werden mithilfe der Matplotlib- und TensorFlow-Bibliotheken gespeichert, um sie nach dem Training zugänglich und vergleichbar zu machen.

```
1 def generate_images(model, test_input, tar, epoch):
2   prediction = model(test_input, training=True)
3   tf.keras.preprocessing.image.save_img(os.path.join(save_path, '
      input_image.png'), test_input[0] * 0.5 + 0.5)
4   tf.keras.preprocessing.image.save_img(os.path.join(save_path, '
      ground_truth.png'), tar[0] * 0.5 + 0.5)
5   tf.keras.preprocessing.image.save_img(os.path.join(save_path, '
      predicted_image.png'), prediction[0] * 0.5 + 0.5)
```

Code-Auszug 5.18: Speicherung der generierten Bilder

Zusätzlich werden die Metriken zur Evaluation der Modelle, wie der Verluste des Generators und des Diskriminators, sowie die Genauigkeit des Diskriminators und der SSIM-Score nach jeder Epoche in einer CSV-Datei festgehalten (Code A.11). Dieser Ansatz ermöglicht eine visuelle Überprüfung des Trainingsverlaufs und erleichtert die Identifikation von möglichen Verbesserungsbereichen. Darüber hinaus ermöglicht die Datei die spätere Erstellung von Verlaufskurven für eine eingehendere Analyse (Code A.13).

Zur Initialisierung der Metriken kommen TensorFlow-Keras-Metriken zum Einsatz:

```
1 gen_loss_metric = tf.keras.metrics.Mean()
2 disc_loss_metric = tf.keras.metrics.Mean()
3 disc_accuracy_metric = tf.keras.metrics.BinaryAccuracy(threshold=0.5)
```

Code-Auszug 5.19: Initialisierung der Metriken

Diese Metriken werden verwendet, um den durchschnittlichen Verlust des Generators, den durchschnittlichen Verlust des Diskriminators und die Genauigkeit des Diskriminators zu erfassen.
Während des Trainingsprozesses wird für jeden Batch von Trainingsbildern der Verlust und die Genauigkeit berechnet. Die Verluste werden durch die oben genannten Metriken erfasst und nach jeder Epoche in die CSV-Datei aufgezeichnet. Die Genauigkeit des Diskriminators wird durch die BinaryAccuracy-Metrik überwacht, wobei ein Schwellenwert von 0,5 festgelegt ist, was in binären Klassifikationsaufgaben typisch ist.

Die Berechnung und Aktualisierung der Metriken erfolgen wie folgt:

```
1 gen_loss, disc_loss, disc_real_output, disc_generated_output =
      train_step(input_image, target, epoch, train_discriminator=True)
2 gen_loss_metric(gen_loss)
3 disc_loss_metric(disc_loss)
4
5 # Echte Bilder : Label 1, generierte Bilder :Label 0
6 real_labels = tf.ones_like(disc_real_output)
7 generated_labels = tf.zeros_like(disc_generated_output)
8 disc_accuracy_metric.update_state(real_labels, disc_real_output)
9 disc_accuracy_metric.update_state(generated_labels,
      disc_generated_output)
```

Code-Auszug 5.20: Aktualisierung der Metriken

Der SSIM-Score kann mithilfe der TensorFlow-Funktion $tf.image.ssim()$ berechnet werden (Code A.12).

Diese systematischen Vorgehensweisen ermöglichen eine umfassende Überprüfung der Modellleistung und bieten visuelle Einblicke in den Fortschritt während des Trainingsprozesses.

6 Evaluation

6.1. Bewertungskriterien

6.1.1. Diskriminator-Verlust

Der Diskriminator-Verlust ist ein wesentliches Maß für die Effektivität des Diskriminators in einem GAN-Modell. Ein niedriger Diskriminator-Verlust bedeutet, dass der Diskriminator erfolgreich zwischen echten und generierten Bildern unterscheiden kann. Im Kontext eines GANs zeigt ein abnehmender Diskriminator-Verlust über die Epochen hinweg, dass der Diskriminator zunehmend besser darin wird, die Authentizität der Bilder zu beurteilen. Ein hoher Diskriminator-Verlust weist hingegen darauf hin, dass das Modell Schwierigkeiten hat, echte von generierten Bildern zu unterscheiden, was ein Indikator für Verbesserungspotenzial im Training oder der Modellarchitektur sein könnte.

6.1.2. Generator-Verlust

Der Generator-Verlust ist ein Indikator für die Fähigkeit des Generators, Bilder zu erzeugen, die vom Diskriminator als echt eingestuft werden. Ein hoher Generator-Verlust deutet darauf hin, dass der Generator die realen Daten noch nicht effektiv nachahmen kann, während ein niedriger Generator-Verlust ein Zeichen dafür ist, dass die generierten Bilder den echten immer ähnlicher werden. Im Verlauf des Trainings sollte der Generator-Verlust tendenziell abnehmen, was darauf hindeutet, dass der Generator lernt, überzeugendere Bilder zu produzieren.

6.1.3. Diskriminator Genauigkeit

Die Diskriminator-Genauigkeit gibt an, wie oft der Diskriminator korrekt zwischen echten und gefälschten Bildern unterscheidet. Eine hohe Genauigkeit bedeutet, dass der Diskriminator effektiv arbeitet, während eine niedrige Genauigkeit auf Probleme bei der Unterscheidung hinweisen kann. Es ist wichtig, ein Gleichgewicht zu finden, denn eine zu hohe Genauigkeit kann darauf hindeuten, dass der Diskriminator die generierten Bilder zu leicht erkennt, was auf ein Problem mit dem Generator hinweisen könnte.
In einem ideal funktionierenden GAN sollte der Diskriminator eine Genauigkeit

von etwa 50% erreichen. Das bedeutet, dass der Diskriminator die echten von dem generierten Bildern nur zufällig unterscheiden bzw. nicht mehr zuverlässig unterscheiden kann, was zeigt dass das GAN gut trainiert wurde.

6.1.4. SSIM-Score (Structural Similarity Index)

Der SSIM-Score ist ein Maß für die visuelle Ähnlichkeit zwischen den generierten Bildern und den entsprechenden Originalbildern. Er bewertet die Bildqualität anhand von Faktoren wie Helligkeit, Kontrast und Struktur. Ein hoher SSIM-Wert deutet darauf in, dass die generierten Bilder den echten Bildern sehr ähnlich sind, was ein Zeichen für eine hohe Bildqualität ist. Ein niedriger SSIM-Wert kann auf deutliche Unterschiede in der visuellen Struktur hinweisen, was Verbesserungsbedarf im Generator-Training oder in der Modellarchitektur signalisiert.

6.2. Ergebnisse und objektive Bewertung

In diesem Abschnitt liegt der Fokus auf der objektiven Bewertung der Leistung des CycleGAN-Algorithmus. Es werden verschiedene Metriken und Kriterien herangezogen, um eine umfassende Bewertung durchzuführen, einschließlich der Verluste der Generatoren und Diskriminatoren bei Variation der Hyperparameter. Darüber hinaus wird eine detaillierte Analyse der Genauigkeit der Diskriminatoren und des SSIM-Scores durchgeführt. Diese Messungen sind sowohl für die genaue Bestimmung der Qualität der generierten Ergebnisse als auch für die Bewertung der Lernfähigkeit der Modelle von Bedeutung.
Die verwendeteten Datensätze für die Evaluierung stammen aus dem Berkeley-Repository. Besondere Beachtung wird dem 'maps'-Datensatz geschenkt, bei dem Transformationen zwischen Kartenansichten und Satellitenbildern durchgeführt wurden. Das Training wurde mehrmals mit verschiedenen Hyperparameter-Konfigurationen durchgeführt, wobei Parameter wie der Lambda-Wert, die Anzahl der Epochen und die Lernrate des Optimizers variiert wurden. Ziel ist es, eine Konfiguration zu finden, bei der die Qualität der erzeugten Bilder nahe an der Qualität des Originals und die Leistung des Diskriminators optimal ist.
Die subjektive Bewertung erfolgt durch die visuelle Beurteilung der generierten Bilder aus dem Testdatensatz hinsichtlich stimmiger Farben, Struktur und Rauschen im Vergleich zu den Originalbildern. Darauf aufbauend erfolgt eine objektive Bewertung mithilfe der definierten Metriken.

6.2.1. Pix2Pix

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.

Durch empirische Tests hat sich herausgestellt das 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).
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.

6.2.2. CycleGAN

Die Datensätze X (Karten) und Y (Satellitenbilder) werden durch die Generatoren $G : X \to Y$ und $F : Y \to X$ sowie die Diskriminatoren D_X und D_Y repräsentiert, die jeweils darauf abzielen, Bilder in den Domänen X und Y zu unterscheiden. Die Hyperparameter basieren zunächst auf den in dem Originalpaper von Zhu ausgewählten Parametern und wurden anschließend durch explorative Anpassungen verfeinert.

Ein herausforderndes Problem, dem sich die Generatoren gegenübersehen, besteht in der Reproduktion feiner und detaillierter Strukturen. Insbesondere fällt auf, dass die Generatoren besser darin sind, Straßenlinien zu erzeugen, was auf die Dominanz von Straßen- und Häusermotiven in den Trainingsdaten zurückzuführen ist. Die Vielfalt in den Bildern, wie Flüsse und Wälder, stellt hingegen eine größere Herausforderung dar. Die Generatoren haben Schwierigkeiten, die Struktur und Farbkodierung dieser Bereiche präzise zu erfassen. Häufig resultiert dies in der Generierung von Flächen mit ähnlicher Farbe wie Straßen, und es werden Versuche unternommen, basierend auf diesen großen Freiflächen kleine Straßen zu generieren.

Die Wahl der Lernrate erwies sich als kritisch, wobei die besten Ergebnisse bei einer Lernrate von $0,0002$ erzielt wurden. Diese Einstellung führte zu den höchsten SSIM-Werten und Diskriminator-Testgenauigkeiten. Insbesondere erzielte der Generator F einen SSIM-Score von 0.592, während der Generator G einen Wert

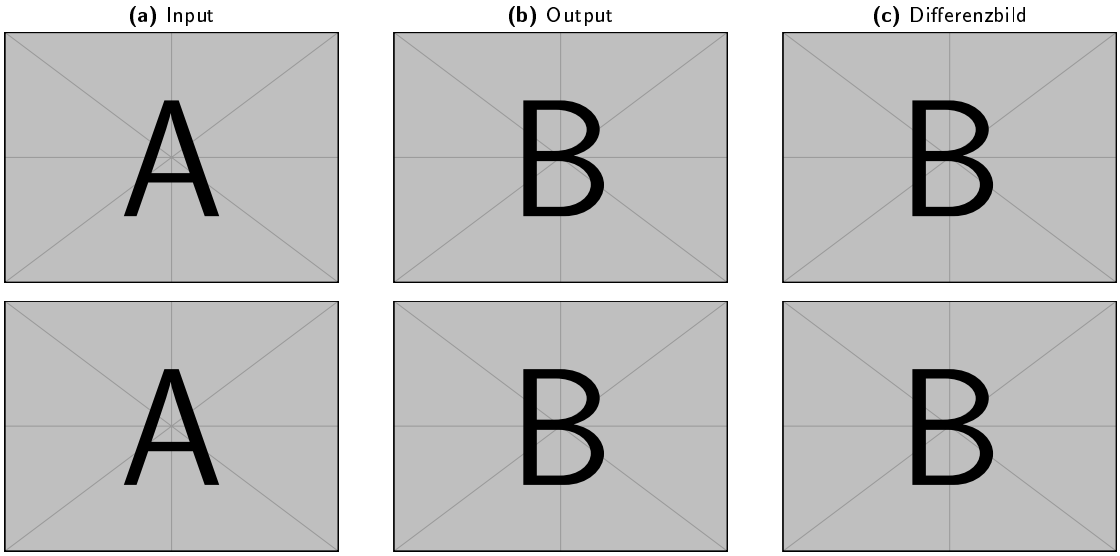
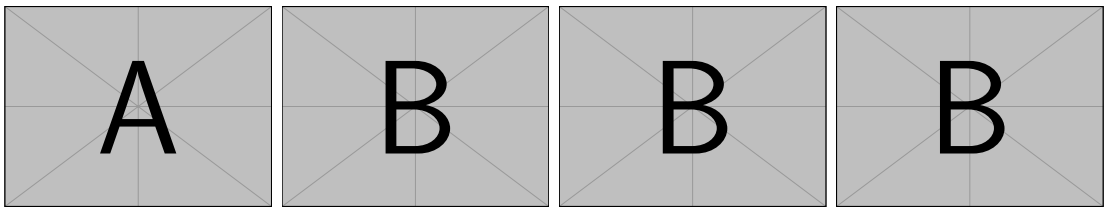


Abbildung 6.1.: Testergebnisse nach dem Training mit $\lambda = 120$, Lernrate $= 0.0002$ und 100 Epochen



(a) (a) Generator-Verlust **(b)** (b) Diskriminator-Metriken **(c)** (c) SSIM-Score **(d)** (d) SSIM-Score Zyklus

Abbildung 6.2.: Metriken nach dem Training mit $\lambda = 120$, Lernrate $= 0.0002$ und 100 Epochen

von 0.198 erreichte. Die Diskriminatoren zeigten mit 50.1% für D_Y und $48,5\%$ für D_X nahezu optimale Genauigkeiten. Trotz der zunächst hoch erscheinenden visuellen Ähnlichkeit zwischen den generierten Satellitenbildern zeigte eine genauere Analyse, dass kleinere Strukturen im generierten Bild fehlten. Dies spiegelte sich in den vergleichsweise niedrigen SSIM-Wert wider.

Ebenso zeigten die Bilder bei einem Gewichtungsfaktor von $\lambda = 120$ die besten Resultate, sowohl basierend auf visueller Beurteilung als auch auf dem SSIM-Score (6.1). Dieser spezifische Wert für λ führte zu generierten Bildern, deren Signal-Rausch-Verhältnis (SNR) entweder besser oder nahezu dem des Originalbildes entspricht. Der SNR-Wert gibt dabei das Verhältnis zwischen dem Signal, repräsentiert durch die relevanten Bildinformationen, und dem Rauschen, repräsentiert durch unerwünschte Störungen, an. In diesem Zusammenhang zeigt ein höherer SNR-Wert eine bessere Qualität und Klarheit der generierten Bilder im Vergleich zu Rauschen.

Trotz dieser Erfolge bleibt das Training instabil, und es besteht die Möglichkeit

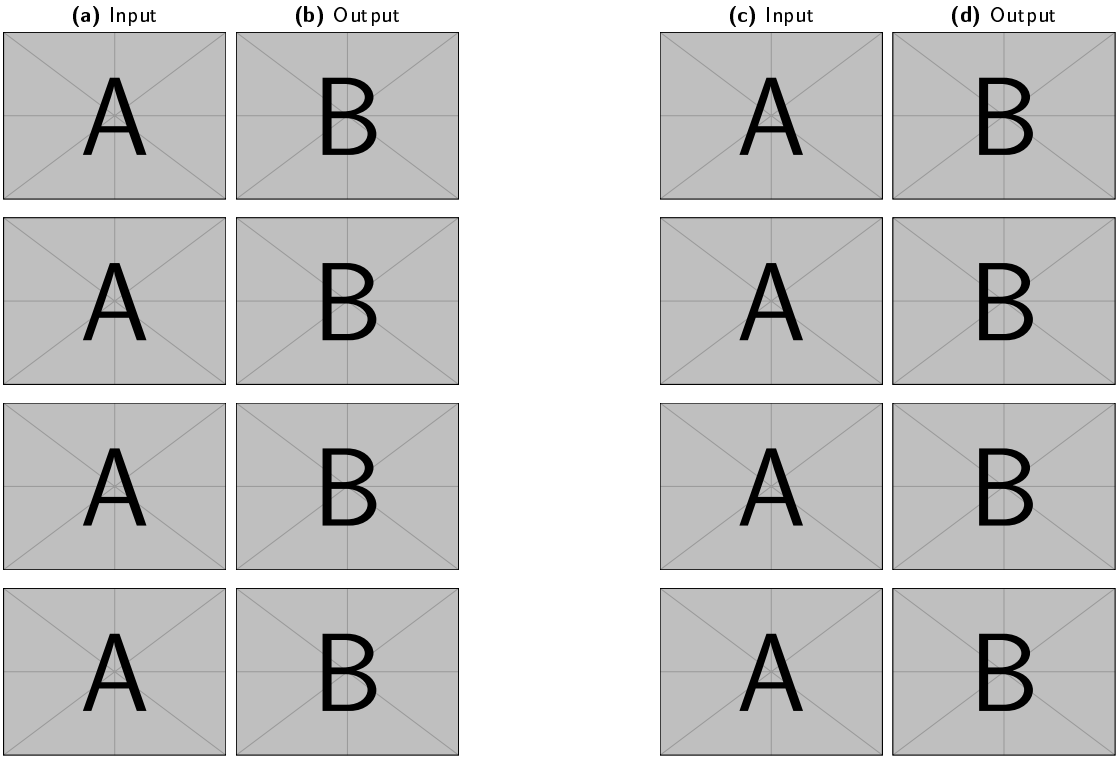


Abbildung 6.3.: Beste Ergebnisse des Trainings

eines Modekollapses. Die Generatoren zeigen Anzeichen eines exponentiellen Verfalls in ihren Verlustkurven, während der SSIM-Score nur langsam ansteigt. Dies deutet darauf hin, dass die Generatoren zwar ihre Verluste minimieren, jedoch möglicherweise nicht die gewünschte visuelle Qualität erreichen (6.2).
Die generierten Bilder weisen bereits zu diesem Zeitpunkt eine grobe visuelle Ähnlichkeit mit der Struktur des zugrunde liegenden Originalbildes auf und zeigen nur minimale Variationen. Allerdings wird durch den SSIM-Score deutlich, dass die Generatoren Schwierigkeiten haben, bedeutende Verbesserungen hinsichtlich feinerer Details und Farbübereinstimmungen zu erzielen. Diese lassen sich auch visuell in den Differenzbilder, die mit Hilfe eines Histogrammsausgleich kontrastverstärkt wurden, erkennen. Helle Bereiche in diesen Differenzbildern stellen dabei größere Unterschiede dar. Dabei sammeln sich helle Bereiche vor allem bei den kleinen pixelweisen Details an, sowie bei unterschiedlichen Farbkodierungen. Ein konkretes Beispiel ist in Abbildung 6.1 zu sehen.
Insbesondere in den frühen Epochen sind die Generatoren nicht in der Lage, Straßenlinien zuverlässig als gerade Linien zu generieren. Diese Schwächen verbessern sich im Laufe längerer Trainingszeiten, wobei bereits nach etwa 300 Epochen relativ konsistente, gerade Linien erzeugt werden.

Die Analyse des SSIM-Scores im Kontext der Zyklusübersetzung offenbart eine exponentielle Zunahme. Diese Beobachtung legt nahe, dass der Zykluskonsistenz-Verlust einen erheblichen Einfluss auf das Training ausübt. Die exponentielle Stei-

Lambda	**Epoche**	**Lernrate**	D_Y **Tr-Acc**	D_X **Tr-Acc**	D_Y **Te-Acc**	D_X **Te-Acc**
120	100	0.000025	0.549	0.494	0.307	0.497
120	100	0.00005	0.537	0.491	0.321	0.489
120	100	0.0001	0.519	0.490	0.418	0.485
120	100	0.0002	0.518	0.495	0.501	0.485
120	100	0.0004	0.516	0.487	0.462	0.496
100	100	0.0002	0.521	0.491	0.527	0.481
140	100	0.0002	0.562	0.492	0.544	0.469
100	300	0.0002	0.51	0.484	0.47	0.676
120	300	0.0002	0.508	0.495	0.468	0.709

Tabelle 6.1.: CycleGAN Ergebnisse unter verschiedenen Hyperparameter Teil1

F **Loss**	G **Loss**	X **SSIM**	Y **SSIM**	X **Z-SSIM**	Y **Z-SSIM**	X **SNR**	Y **SNR**
53.113	22.556	0.545	0.191	0.895	0.831	-2.991	0.228
44.626	20.417	0.541	0.191	0.916	0.836	-0.589	1.071
44.376	21.943	0.580	0.194	0.935	0.842	-4.604	1.367
42.489	22.373	0.592	0.198	0.921	0.796	-5.308	0.851
46.972	28.175	0.585	0.209	0.856	0.602		
34.973	18.837	0.551	0.194	0.93	0.83	-3.292	1.261
13.048	6.842	0.510	0.179	0.513	0.409	-2.087	1.005
34.881	20.899	0.577	0.191	0.931	0.756	-2.705	1.373
46.652	26.360	0.585	0.207	0.901	0.707	1.248	-3.409

Tabelle 6.2.: CycleGAN Ergebnisse unter verschiedenen Hyperparameter Teil2

gerung des SSIM-Scores deutet darauf hin, dass die Konsistenz zwischen den Original- und zurückübersetzten Bildern im Laufe der Zeit kontinuierlich zunimmt. Dies könnte auf eine fortlaufende Verbesserung der Generatoren hinsichtlich ihrer Fähigkeit zur Zyklusübersetzung hindeuten. Insgesamt erreichen die SSIM-Scores für die Zyklusübersetzung Werte von 80% für Y und 90% für X (6.2).
Im Kontrast dazu bleibt die Genauigkeit der Diskriminatoren für Trainingsdaten stabil bei etwa 50%. Jedoch zeigen sich erhebliche Schwankungen je nach Epoche für die Testdaten, insbesondere für D_Y. Dies legt nahe, dass die Diskriminatoren möglicherweise Schwierigkeiten haben, mit neuen, nicht trainierten Daten umzugehen. Diese Schwankungen könnten auf Überanpassung oder eine mangelnde Generalisierungsfähigkeit des Modells hinweisen, trotz der implementierten Maßnahmen wie dem Hinzufügen von Rauschen und Dropout-Schichten (6.2).

Die Laufzeit des Trainings spiegelt die Komplexität der Modelle wider, insbesondere auf der GPU, wo jede Epoche im Durchschnitt etwa 2 Minuten dauert. Im Vergleich dazu benötigt die CPU, selbst nach einer Reduzierung der Residualblöcke auf einen Block, rund 6 Minuten pro Epoche. Die Nutzung von Instanznormali-

sierung trägt zur beschleunigten Konvergenz und Verbesserung der Laufzeit bei. Zudem zeigt sich, dass die Verwendung von Residualblöcken eine bessere Laufzeit ermöglicht im Vergleich zu Generatoren mit zusätzlichen Schichten.

Insgesamt zeigt die Evaluation, dass das CycleGAN-Modell trotz einiger Herausforderungen in der Generierung von detaillierten Strukturen und der Generalisierung auf neue Daten vielversprechende Ergebnisse erzielt.

6.3. 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.

A
Anhang - Code

Dieser Abschnitt enthält die wichtigsten Funktionen, die in der Implementierung verwendet werden. Hier finden sich auch die fehlenden Funktionen aus dem Kapitel 'Lösungsbeschreibung'.

Datenvorverarbeitung

```
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 A.1: Lesen eines Bildes (CycleGAN Implementierung)

Pix2Pix

```python
def train_step(input_image, target, epoch, train_discriminator=True):
    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        gen_output = generator(input_image, training=True)

        disc_real_output = discriminator([input_image, target], training
            =True)
        disc_generated_output = discriminator([input_image,
            gen_output], training=True)

        gen_total_loss, gen_gan_loss, gen_l1_loss = generator_loss(
            disc_generated_output, gen_output, target)
        disc_loss = discriminator_loss(disc_real_output,
            disc_generated_output)

    generator_gradients = gen_tape.gradient(gen_total_loss, generator.
        trainable_variables)
    generator_optimizer.apply_gradients(zip(generator_gradients,
        generator.trainable_variables))

    if train_discriminator:
        discriminator_gradients = disc_tape.gradient(disc_loss,
            discriminator.trainable_variables)
        discriminator_optimizer.apply_gradients(zip(
            discriminator_gradients, discriminator.trainable_variables))

    return gen_total_loss, disc_real_output, disc_real_output,
        disc_generated_output
```

Code-Auszug A.2: Pix2Pix Trainingsfunktion

CycleGAN

```
1  def residual_block(x, filters):
2      shortcut = x
3      x = tf.keras.layers.Conv2D(filters, kernel_size=3, strides=1,
           padding='same')(x)
4      x = InstanceNormalization(axis=-1)(x)
5      x = tf.keras.layers.LeakyReLU()(x)
6      x = tf.keras.layers.Conv2D(filters, kernel_size=3, strides=1,
           padding='same')(x)
7      x = InstanceNormalization(axis=-1)(x)
8      x = tf.keras.layers.Concatenate()([x, shortcut])
9      return x
```

Code-Auszug A.3: Residualblock Implementierung

```
1  def convolutional_layer(x, filters, kernel_size=3, strides=2):
2      x = tf.keras.layers.Conv2D(filters, kernel_size=kernel_size, strides
           =strides, padding='same')(x)
3      InstanceNormalization(axis=-1)(x)
4      x = tf.keras.layers.LeakyReLU()(x)
5      return x
```

Code-Auszug A.4: Convolutional Block in CycleGAN

```
1 def t_convolutional_layer(x, filters , kernel_size=3, strides=2):
2     x = tf.keras.layers.Conv2DTranspose(filters, kernel_size=kernel_size
          , strides=strides, padding='same')(x)
3     InstanceNormalization(axis=-1)(x)
4     x = tf.keras.layers.LeakyReLU()(x)
5     return x
```

Code-Auszug A.5: Transpose Convolutional Block in CycleGAN

```
1 def generator_adversarial_loss(generated):
2     return loss_obj(tf.ones_like(generated), generated)
```

Code-Auszug A.6: Adversarieller Verlust des Generators in CycleGAN

```
1 def cycle_loss(real_image, cycled_image):
2   loss1 = tf.reduce_mean(tf.abs(real_image - cycled_image))
3   return LAMBDA * loss1
```

Code-Auszug A.7: Zykluskonsistenz Verlust in CycleGAN

Training und Hyperparameter

```
1  def identity_loss(real_image, same_image):
2    loss = tf.reduce_mean(tf.abs(real_image - same_image))
3    return LAMBDA * 0.5 * loss
```

Code-Auszug A.8: Identitätsverlust in CycleGAN

```
def train_step(real_x, real_y):
  with tf.GradientTape(persistent=True) as tape:
      fake_y = generator_g(real_x, training=True)
      cycled_x = generator_f(fake_y, training=True)

      fake_x = generator_f(real_y, training=True)
      cycled_y = generator_g(fake_x, training=True)

      same_x = generator_f(real_x, training=True)
      same_y = generator_g(real_y, training=True)

      disc_real_x = discriminator_x(real_x, training=True)
      disc_real_y = discriminator_y(real_y, training=True)
      disc_fake_x = discriminator_x(fake_x, training=True)
      disc_fake_y = discriminator_y(fake_y, training=True)

      total_gen_g_loss = generator_loss(real_x, real_y, cycled_x,
          cycled_y, disc_fake_y, same_y)
      total_gen_f_loss = generator_loss(real_x, real_y, cycled_x,
          cycled_y, disc_fake_x, same_x)

      disc_x_loss = discriminator_loss(disc_real_x, disc_fake_x)
      disc_y_loss = discriminator_loss(disc_real_y, disc_fake_y)
```

Code-Auszug A.9: CycleGAN Trainingsfunktion, Teil 1

```
    # Calculate the gradients for generator and discriminator
    generator_g_gradients = tape.gradient(total_gen_g_loss,
                                          generator_g.trainable_variables)
    generator_f_gradients = tape.gradient(total_gen_f_loss,
                                          generator_f.trainable_variables)
    discriminator_x_gradients = tape.gradient(disc_x_loss,
                                              discriminator_x.
                                                  trainable_variables)
    discriminator_y_gradients = tape.gradient(disc_y_loss,
                                              discriminator_y.
                                                  trainable_variables)

    # Apply the gradients to the optimizer
    generator_g_optimizer.apply_gradients(zip(generator_g_gradients,
                                              generator_g.
                                                  trainable_variables))
    generator_f_optimizer.apply_gradients(zip(generator_f_gradients,
                                              generator_f.
                                                  trainable_variables))
    discriminator_optimizer_x.apply_gradients(zip(
        discriminator_x_gradients,
                                              discriminator_x.
                                                  trainable_variables))
    discriminator_optimizer_y.apply_gradients(zip(
        discriminator_y_gradients,
                                              discriminator_y.
                                                  trainable_variables))
    return total_gen_f_loss, total_gen_g_loss, disc_y_loss, disc_x_loss,
         fake_x, fake_y, disc_real_x, disc_real_y, disc_fake_x,
        disc_fake_y
```

Code-Auszug A.10: CycleGAN Trainingsfunktion, Teil 2

```
def save_losses(epoch, gen_loss, disc_loss, disc_accuracy, ssim):
  with open(losses_path, 'a') as f:
    writer = csv.writer(f)
    writer.writerow([epoch, gen_loss.result().numpy(), disc_loss.result()
        .numpy(), disc_accuracy, ssim])
```

Code-Auszug A.11: Speicherung der Metriken in CSV-Datei

```python
def calculate_ssim(generator, dataset, num_samples=10):
    ssim_values = []
    for input_image, target_image in dataset.take(num_samples):
        prediction = generator(input_image, training=False)
        ssim_value = tf.image.ssim(prediction, target_image, max_val
            =2.0)
        ssim_values.append(ssim_value)
    return tf.reduce_mean(ssim_values)
```

Code-Auszug A.12: Berechnung des SSIM-Score

```python
def plot_curve(name, epoch, col2, col_name, num_curves=1):
    path = f"./satelite/try1/Curves/{name}.jpg"
    if not os.path.exists("./satelite/try1/Curves/"):
        os.makedirs("./satelite/try1/Curves/")
    for i in range(num_curves):
        plt.plot(epoch, col2.iloc[:, i], label=col_name[i])
    plt.xlabel('Epoche')
    plt.ylabel(name)
    plt.title(f'Epoche vs {name}')
    plt.legend()
    plt.savefig(path)
    plt.show()

def save_curves():
    df = pd.read_csv('./satelite/try1/losses.csv')
    epoche = df.iloc[:, 0]
    gen_loss = df.iloc[:, [1, 2]]
    plot_curve("Generator_Verlust", epoche, gen_loss, ["
        Generator_F_Verlust", "Generator_G_Verlust"], 2)
```

Code-Auszug A.13: Ausschnitt zur Erstellung einer Verlaufskurve (CycleGAN Implementierung)

B
Anhang - Modelle

In diesem Abschnitt werden die Modelle der Generatoren und Diskriminatoren aufgelistet.

```
1  def Generator():
2    inputs = tf.keras.layers.Input(shape=[256, 256, 3])
3
4    down_stack = [
5      downsample(64, 4, apply_batchnorm=True), # (batch_size, 128, 128,
           64)
6      downsample(128, 4), # (batch_size, 64, 64, 128)
7      downsample(256, 4), # (batch_size, 32, 32, 256)
8      downsample(512, 4), # (batch_size, 16, 16, 512)
9      downsample(512, 4), # (batch_size, 8, 8, 512)
10     downsample(512, 4), # (batch_size, 4, 4, 512)
11     downsample(512, 4), # (batch_size, 2, 2, 512)
12     downsample(512, 4), # (batch_size, 1, 1, 512)
13   ]
14
15   up_stack = [
16     upsample(512, 4, apply_dropout=True), # (batch_size, 2, 2, 1024)
17     upsample(512, 4, apply_dropout=True), # (batch_size, 4, 4, 1024)
18     upsample(512, 4, apply_dropout=True), # (batch_size, 8, 8, 1024)
19     upsample(512, 4),  # (batch_size, 16, 16, 1024)
20     upsample(256, 4),  # (batch_size, 32, 32, 512)
21     upsample(128, 4),  # (batch_size, 64, 64, 256)
22     upsample(64, 4),  # (batch_size, 128, 128, 128)
23   ]
```

Code-Auszug B.1: Pix2Pix Generator in Tensorflow Teil 1

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

Code-Auszug B.2: Pix2Pix Generator in Tensorflow Teil 2

```python
def Discriminator():
    initializer = tf.random_normal_initializer(0., 0.02)

    inp = tf.keras.layers.Input(shape=[256, 256, 3], name='input_image')
    tar = tf.keras.layers.Input(shape=[256, 256, 3], name='target_image')

    x = tf.keras.layers.concatenate([inp, tar])  # (batch_size, 256, 256, channels*2)

    down1 = downsample(64, 4, False)(x) # (batch_size, 128, 128, 64)
    down2 = downsample(128, 4)(down1) # (batch_size, 64, 64, 128)
    down3 = downsample(256, 4)(down2) # (batch_size, 32, 32, 256)

    zero_pad1 = tf.keras.layers.ZeroPadding2D()(down3) # (batch_size, 34, 34, 256)
    conv = tf.keras.layers.Conv2D(512, 4, strides=1,
    kernel_initializer=initializer, use_bias=False)(zero_pad1) # (batch_size, 31, 31, 512)

    batchnorm1 = tf.keras.layers.BatchNormalization()(conv)

    leaky_relu = tf.keras.layers.LeakyReLU()(batchnorm1)

    zero_pad2 = tf.keras.layers.ZeroPadding2D()(leaky_relu) # (batch_size, 33, 33, 512)

    last = tf.keras.layers.Conv2D(1, 4, strides=1,
    kernel_initializer=initializer)(zero_pad2) # (batch_size, 30, 30, 1)

    return tf.keras.Model(inputs=[inp, tar], outputs=last)
```

Code-Auszug B.3: Pix2Pix Diskriminator in Tensorflow

```
def Generator():
    inputs = tf.keras.layers.Input(shape=[None,None,3])

    # Layer 1
    x = layers.Conv2D(64, 7, strides=1, padding='same')(inputs)
    x = layers.BatchNormalization()(x)
    x = layers.Activation('relu')(x)

    # Layer 2+ 3
    x = convolutional_layer(x, 128, 3,2)
    x = convolutional_layer(x, 256, 3, 2)

    # Residual Blocks
    for _ in range(6):
        x = residual_block(x, 256)

    # Layer 10 + 11
    x = t_convolutional_layer(x, 128, 3, 2)
    x = t_convolutional_layer(x, 64, 3, 2)

    # Dropout
    x = tf.keras.layers.Dropout(0.5)(x)

    # Layer 12
    outputs = layers.Conv2D(3, 7, strides=1, padding='same',
        activation='tanh')(x)

    return tf.keras.Model(inputs=inputs, outputs=outputs)
```

Code-Auszug B.4: CycleGAN Generator in Tensorflow

```python
def Discriminator():
    model = tf.keras.Sequential()

    # Layer 1
    model.add(layers.Conv2D(64, 4, strides=2, padding='same',
        input_shape=(256,256,3)))
    model.add(layers.BatchNormalization())
    model.add(layers.Activation('relu'))

    # Layer 2
    model.add(layers.Conv2D(128, 4, strides=2, padding='same'))
    model.add(layers.BatchNormalization())
    model.add(layers.Activation('relu'))

    # Layer 3
    model.add(layers.Conv2D(256, 4, strides=2, padding='same'))
    model.add(layers.BatchNormalization())
    model.add(layers.Activation('relu'))

    # Layer 4
    model.add(layers.Conv2D(512, 4, strides=2, padding='same'))
    model.add(layers.BatchNormalization())
    model.add(layers.Activation('relu'))

    # Layer 5
    model.add(layers.Conv2D(1, 4, strides=1, padding='same'))
    model.add(layers.BatchNormalization())
    model.add(layers.Activation('sigmoid'))

    # Dropout
    model.add(tf.keras.layers.Dropout(0.5))
    return model
```

Code-Auszug B.5: CycleGAN Diskriminator in Tensorflow

Literaturverzeichnis

[.2010] *Proceedings of the 27th International Conference on International Conference on Machine Learning*. Madison, WI, USA : Omnipress, 2010 (ICML'10). – ISBN 9781605589077

[.2017] *Proceedings of the 34th International Conference on Machine Learning - Volume 70*. JMLR.org, 2017 (ICML'17)

[.2019] *Proceedings of The 7th International Conference on Intelligent Systems and Image Processing 2019*. The Institute of Industrial Application Engineers, 2019 . – ISBN 9784907220198

[ACB17] Arjovsky, Martin ; Chintala, Soumith ; Bottou, Léon: Wasserstein Generative Adversarial Networks. In: *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, JMLR.org, 2017 (ICML'17), S. 214–223

[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.jjimei.2020.100004. – DOI 10.1016/j.jjimei.2020.100004. – ISSN 26670968

[Ara22] Aram You, Jin Kuk Kim, Ik Hee Ryu, Tae Keun Yoo: Application of generative adversarialnetworks (GAN) for ophthalmology imagedomains: a survey. (2022)

[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

[CZS] Chu, Casey ; Zhmoginov, Andrey ; Sandler, Mark: *CycleGAN, a Master of Steganography*

[EO] Eckerli, Florian ; Osterrieder, Joerg: *Generative Adversarial Networks in finance: an overview*

[Haz21] Hazem Abdelmotaal, Ahmed A. Abdou, Ahmed F. Omar, Dalia Mohamed El-Sebaity, Khaled Abdelazeem: Pix2pix Conditional Generative Adversarial Networks forScheimpflug Camera Color-Coded Corneal TomographyImage Generation. (2021)

[HB] Huang, Xun ; Belongie, Serge: *Arbitrary Style Transfer in Real-time with Adaptive Instance Normalization*

[HGAPC] Hitaj, Briland ; Gasti, Paolo ; Ateniese, Giuseppe ; Perez-Cruz, Fernando: *PassGAN: A Deep Learning Approach for Password Guessing*. http://arxiv.org/pdf/1709.00440v3

[HHYY20] Hong, Yongjun ; Hwang, Uiwon ; Yoo, Jaeyoon ; Yoon, Sungroh: How Generative Adversarial Networks and Their Variants Work. In: *ACM Computing Surveys* 52 (2020), Nr. 1, S. 1–43. http://dx.doi.org/10.1145/3301282. – DOI 10.1145/3301282. – ISSN 0360–0300

[HZRS] He, Kaiming ; Zhang, Xiangyu ; Ren, Shaoqing ; Sun, Jian: *Deep Residual Learning for Image Recognition*

[Ian14] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio: Generative Adversarial Nets. (2014)

[JZJ+20] Jain, Deepak K. ; Zareapoor, Masoumeh ; Jain, Rachna ; Kathuria, Abhishek ; Bachhety, Shivam: GAN-Poser: an improvised bidirectional GAN model for human motion prediction. In: *Neural Computing and Applications* 32 (2020), Nr. 18, S. 14579–14591

[KB] Kingma, Diederik P. ; Ba, Jimmy: *Adam: A Method for Stochastic Optimization*

[KSH17] Krizhevsky, Alex ; Sutskever, Ilya ; Hinton, Geoffrey E.: ImageNet classification with deep convolutional neural networks. In: *Communications of the ACM* 60 (2017), Nr. 6, S. 84–90. http://dx.doi.org/10.1145/3065386. – DOI 10.1145/3065386. – ISSN 0001–0782

[LCS+21] Liu, Yanxia ; Chen, Anni ; Shi, Hongyu ; Huang, Sijuan ; Zheng, Wanjia ; Liu, Zhiqiang ; Zhang, Qin ; Yang, Xin: CT synthesis from MRI using multi-cycle GAN for head-and-neck radiation therapy. In: *Computerized medical imaging and graphics : the official journal of the Computerized Medical Imaging Society* 91 (2021), S. 101953. http://dx.doi.org/10.1016/j.compmedimag.2021.101953. – DOI 10.1016/j.compmedimag.2021.101953

[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*

[MO] Mirza, Mehdi ; Osindero, Simon: *Conditional Generative Adversarial Nets*

[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

[ON] O'Shea, Keiron ; Nash, Ryan: *An Introduction to Convolutional Neural Networks*

[PJTA] Phillip Isola ; Jun-Yan Zhu ; Tinghui Zhou ; Alexei A. Efros: Image-to-Image Translation with Conditional Adversarial Networks

[PYY+19] Pan, Zhaoqing ; Yu, Weijie ; Yi, Xiaokai ; Khan, Asifullah ; Yuan, Feng ; Zheng, Yuhui: Recent Progress on Generative Adversarial Networks (GANs): A Survey. In: *IEEE Access* 7 (2019), S. 36322–36333. http://dx.doi.org/10.1109/ACCESS.2019.2905015. – DOI 10.1109/ACCESS.2019.2905015

[RMC] Radford, Alec ; Metz, Luke ; Chintala, Soumith: *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*

[SC21] Saxena, Divya ; Cao, Jiannong: Generative adversarial networks (GANs) challenges, solutions, and future directions. In: *ACM Computing Surveys (CSUR)* 54 (2021), Nr. 3, S. 1–42

[SGZ+] Salimans, Tim ; Goodfellow, Ian ; Zaremba, Wojciech ; Cheung, Vicki ; Radford, Alec ; Chen, Xi: *Improved Techniques for Training GANs*

[SSA17] Sharma, Sagar ; Sharma, Simone ; Athaiya, Anidhya: Activation functions in neural networks. In: *Towards Data Sci* 6 (2017), Nr. 12, S. 310–316

[SSJ22] Sharma, Neha ; Sharma, Reecha ; Jindal, Neeru: Comparative analysis of CycleGAN and AttentionGAN on face aging application. In: *Sādhanā* 47 (2022), Nr. 1. http://dx.doi.org/10.1007/s12046-022-01807-4. – DOI 10.1007/s12046–022–01807–4. – ISSN 0256–2499

[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*

[YNDT18] Yamashita, Rikiya ; Nishio, Mizuho ; Do, Richard Kinh G. ; Togashi, Kaori: Convolutional neural networks: an overview and application in radiology. In: *Insights into imaging* 9 (2018), Nr. 4, S. 611–629. http://dx.doi.org/10.1007/s13244-018-0639-9. – DOI 10.1007/s13244–018–0639–9. – ISSN 1869–4101

[YWB19] Yi, Xin ; Walia, Ekta ; Babyn, Paul: Generative adversarial network in medical imaging: A review. In: *Medical image analysis* 58 (2019), S. 101552. http://dx.doi.org/10.1016/j.media.2019.101552. – DOI 10.1016/j.media.2019.101552

[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*

[ZZS+18] Zheng, Yu-Jun ; Zhou, Xiao-Han ; Sheng, Wei-Guo ; Xue, Yu ; Chen, Sheng-Yong: Generative adversarial network based telecom fraud detection at the receiving bank. In: *Neural networks : the official journal of the International Neural Network Society* 102 (2018), S. 78–86. http://dx.doi.org/10.1016/j.neunet.2018.02.015. – DOI 10.1016/j.neunet.2018.02.015

Abbildungsverzeichnis

Tabellenverzeichnis

Code-Auszugs-Verzeichnis

Glossar

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

Arbeitsverteilung

Teilnehmer 1: Elisa Du

Inhalte:

Grundlagen - Kapitel (exklusiv Pix2Pix)
Problembeschreibung - Kapitel
Lösungsbeschreibung - Kapitel (exklusiv Implementierung der Pix2PixGAN-Architektur)

Teilnehmer 2: Marcel Hoffmann

Inhalte:

Pix2Pix in Grundlagen - Kapitel
Implementierung der Pix2PixGAN-Architektur in Lösungsbeschreibung - Kapitel

Arbeitsverteilung

Teilnehmer 1: Elisa Du

Inhalte:

Abstract/Zusammenfassung
Grundlagen - Kapitel (exklusiv Pix2Pix)
Literaturreview
Problembeschreibung - Kapitel
Lösungsbeschreibung - Kapitel (exklusiv Implementierung der Pix2PixGAN-Architektur)

Teilnehmer 2: Marcel Hoffmann

Inhalte:

Einleitung
Pix2Pix in Grundlagen - Kapitel
Implementierung der Pix2PixGAN-Architektur in Lösungsbeschreibung - Kapitel