Technische Hochschule
Brandenburg
University of
Applied Sciences

Bachelorarbeit am Fachbereich
Informatik und Medien

Umsetzung und Vergleich von GANs (Generative Adversarial Networks) zur Generierung von Bildern menschlicher Gesichter

Eingereicht von:	Nawid Shadab
Erster Gutachter:	Dipl.-Inform. Ingo Boersch
Zweiter Gutachter:	Prof. Dr.-Ing. Jochen Heinsohn
Abgabetermin :	02.02.2022

Selbständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit zum Thema

**Umsetzung und Vergleich von GANs (Generative Adversarial Networks)
zur Generierung von Bildernmenschlicher Gesichter**

selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Brandenburg, 02.02.2022 Nawid Shadab

Danksagungen

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich während der Anfertigung dieser Bachelorarbeit unterstützt und motiviert haben.

Zuerst gebührt mein Dank Herrn Dipl.-Inform. Ingo Boersch und Herrn Prof. Dr. -Ing. Jochen Heinsohn, die meine Bachelorarbeit betreut und begutachtet haben. Ich bedanke mich herzlich für die hilfreichen Anregungen und die konstruktive Kritik bei der Erstellung dieser Arbeit.

Ebenfalls möchte ich mich bei Frau Dr. Petra Hoffmann für die sprachliche Konsultation und Unterstützung bedanken.

Abschließend möchte ich mich bei meinen Eltern und meiner Familie bedanken, die mir mein Studium durch ihre Unterstützung ermöglicht haben.

Nawid Shadab
Potsdam, 02.02.2022

Zusammenfassung

Generative Adversarial Networks (GANs) sind ein neuer Ansatz im Bereich Deep Neural Network. Diese Netzwerke haben nachgewiesen, dass sie bei der Erzeugung realistischer Bilder aus der realen Welt erfolgreich sind. GANs bestehen aus zwei unterschiedlichen Modellen, dem sogenannten Discriminator und dem Generator. Der Discriminator lernt, die Bilder als echt oder gefälscht zu klassifizieren. Der Generator nimmt ein zufälliges Rauschen als Eingabe und lernt, echte Bilder zu erzeugen. Das Generator- und Discriminator-Modell werden gemeinsam trainiert, um ein Gleichgewicht zu erreichen. In der vorliegenden Arbeit wurden zunächst die relevanten Grundlagen zu neuronalen Netzen und GANs erläutert. Dann wurde das Konzept von drei GAN-Modellen, nämlich Standard-GAN (SGAN), Deep Convolutional GAN (DCGAN) und Wasser Stein GAN with Gradient Penalty (WGAN-GP) näher betrachtet und diese Modelle wurden unter Verwendung von Pytorch implementiert. Die Modelle wurden auf dem CelebA Datensatz trainiert. Schließlich wurde versucht, diese GAN-Modelle anhand der Qualität der von ihnen erzeugten Bilder zu vergleichen. Dazu wurden die durch SGAN, DCGAN und WGAN-GP generierten Bilder in zwei Experimenten quantitativ (mittels FID-Metrik) und qualitativ (objektive Beobachtung) ausgewertet. In den Experimenten wurde festgestellt, dass das DCGAN-Modell bessere Bilder als SGAN und WGAN-GP erzeugt. Außerdem wurde festgestellt, dass der Wasser Stein Loss with Gradient Penalty (WGP-Loss) keinen großen Einfluss auf die Qualität der erzeugten Bilder hat.

Inhaltsverzeichnis

1 Einleitung

Generative Adversarial Networks (GANs) [Goodfellow et al., 2014] [1] sind eine Unterklasse von generativen Modellen, die die KI revolutioniert haben und nun in der Lage sind, qualitativ hochwertige Bilder zu generieren. Dieses Netz besteht aus zwei konkurrierenden Netzen, dem Generator und dem Discriminator. Der Generator versucht, ein Mapping von einer Rausch-Verteilung auf die reale Daten-Verteilung zu lernen. Der Discriminator unterscheidet zwischen echten Datenproben und den vom Generator erzeugten Proben. Diese beiden Netzwerke konkurrieren gegeneinander in einem Minimax-Spiel mit zwei Spielern. Der Generator versucht, möglichst gute Proben zu erzeugen, damit der Discriminator sie nicht von echten Proben unterscheiden kann. Auf der anderen Seite versucht der Discriminator, seine Fähigkeit zur Unterscheidung zwischen echten und gefälschten Daten zu verbessern, indem er nur die echten Daten als echt klassifiziert. Der Generator hat keinen Zugang zu den echten Daten, er lernt nur aus dem Feedback des Discriminator.

Seit 2014, wo das erste GAN-Modell vorgestellt wurde, hat sich dieses Netzwerk stark verbessert und hat verschiedene Anwendungsmöglichkeiten gefunden. Derzeit sind die neuen GAN-Ansätze wie StyleGAN3 [2] in der Lage, sehr realistische und qualitativ hochwertige Bilder zu erzeugen.

Das Ziel dieser Arbeit besteht darin, die Fähigkeit von drei GAN-Modellen (SGAN, DCGAN und WGAN-GP) mit unterschiedlichen Strukturen und unterschiedlichen Verlustfunktionen bei der Bildsynthese (Bilderzeugung) zu vergleichen. Außerdem wird mit der vorliegenden Arbeit versucht, einen Überblick über verschiedene GAN-Modelle (SGAN, DCGAN und WGAN-GP) und deren Komponenten und Funktionalitäten zu schaffen. Darüber hinaus werden diese Modelle und ihre Fähigkeiten zur Generierung neuer Bilder (menschliche Gesichter) nach dem Training mit dem CelebA-Datensatz verglichen. Die Qualität der von diesen Modellen (SGAN, DCGAN und WGAN-GP) erzeugten Bilder wird qualitativ und quantitativ überprüft. Die quantitative Bewertung erfolgt unter Verwendung der Fréchet Inception Distance (FID) [3] Metrik, die eines der gängigsten Maßstäbe zur Bewertung von GANs ist [4]. Die qualitative Evaluierungsmethode erfolgt objektiv und auf menschlicher Basis. Hierfür wird das Konzept der Rapid Scene Categorization [4] Methode verwendet.

2 Grundlagen zu neuronalen Netzen

Die neuronalen Netze, die auch als künstliche neuronale Netze (KNN) oder simulierte neuronale Netze (SNN) bezeichnet werden, sind eine beliebte Technik des maschinellen Lernens, die den Lernmechanismus in biologischen neuronalen Netzen simulieren [5].
Künstliche neuronale Netze (KNN) werden von Computerprogrammen zur Erkennung von Patterns und zur Lösung allgemeiner Probleme in den Bereichen KI, maschinelles Lernen und Deep Learning eingesetzt. Sie bilden den Kern der Deep-Learning-Algorithmen [6].

In dieser Bachelorarbeit wird der Begriff „neuronale Netze“ verwendet, um sich auf künstliche neuronale Netze (KNN) und nicht auf biologische Netze zu beziehen.

In diesem Abschnitt beschränken wir uns auf die Grundlagen und die Einordnung einiger Begrifflichkeiten, die das Verständnis des Konzepts der GANs (Generative Adversarial Networks) erleichtern.

2.1 Struktur neuronaler Netze

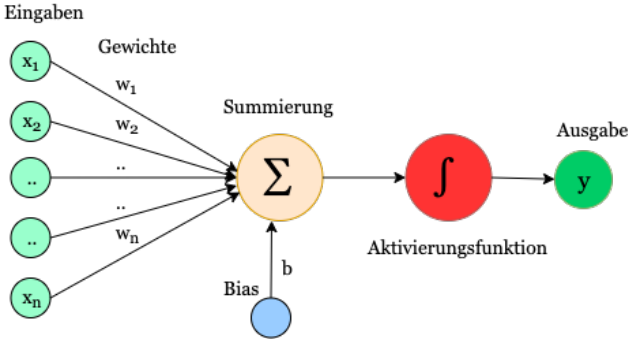
Der Hauptbaustein für neuronale Netze sind künstliche Neuronen. Ein künstliches Neuron, auch *Perzeptron* genannt, ist eine einfache Recheneinheit, die ein gewichtetes Signal als Eingabe erhält und mit Hilfe einer Aktivierungsfunktion ein Ausgangssignal erzeugt [7].
Ein **Perzeptron** ist das einfachste und ein einschichtiges neuronales Netz [5]. Ein **mehrschichtiges Perzeptron** wird als neuronale Netze bezeichnet (siehe Abbildung 2.1).

Die Funktionalität von neuronalen Netzen
Die neuronalen Netze funktionieren auf die gleiche Weise wie das Perzeptron. Um zu verstehen, wie die neuronale Netze funktionieren, sollte man lernen, wie ein Perzeptron funktioniert. Ein Perzeptron besteht aus vier Hauptteilen: *Eingabewerte, Gewichte und Bias, Netz-summe und eine Aktivierungsfunktion* [8].
Gleichung 2.1 stellt die mathematische Berechnung hinter einem Perzeptron dar [6].

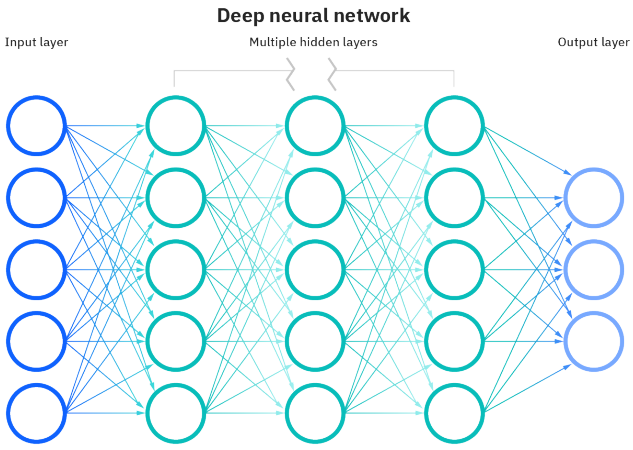
$$\sum_{i=1}^{n} w_i x_i + bias = w_1 x_1 + w_2 x_2 + w_3 x_3 + bias \tag{2.1}$$

Nachdem die Eingabeschicht festgelegt wurde, werden die Gewichte zugewiesen. Diese Gewichte tragen dazu bei, die Wichtigkeit einer bestimmten Variablen zu bestimmen, wobei größere Variablen im Vergleich zu anderen Eingaben einen größeren Einfluss auf die Ausgabe haben. Alle Eingaben werden dann mit ihren jeweiligen Gewichten multipliziert und dann

summiert. Anschließend wird die Ausgabe durch eine Aktivierungsfunktion verarbeitet, die die Ausgabe bestimmt. Wenn die Ausgabe einen bestimmten Schwellenwert übertrifft, wird der Knoten abgefeuert (oder aktiviert) und die Daten werden an die nächste Schicht des Netzes weitergeleitet. Dies führt dazu, dass die Ausgabe eines Knotens zur Eingabe des nächsten Knotens wird. Dieser Prozess der Weitergabe von Daten von einer Schicht an die nächste Schicht definiert dieses neuronale Netz als Feedforward-Netz.
Mit dem Bias-Wert kann die Aktivierungsfunktion nach links oder rechts verschoben werden, um eine bessere Anpassung an die Daten zu erreichen. Der Bias-Wert beeinflusst nur die Ausgabe-werte, er beeinflusst nicht die eigentlichen Eingabedaten [9].



(a) Perzeptron [1]



(b) Mehrschichtiges Perzeptronen [6]

Abbildung 2.1: Struktur neuronaler Netze

[1]Quelle: Eigene Darstellung

2.2 Aktivierungsfunktionen

Aktivierungsfunktionen, die auch *Transferfunktionen* genannt werden, begrenzen die Ausgabe des Neurons in einem Wertebereich [10] und führen eine Nicht-Linearität in das neuronale Netz ein. Ohne eine nichtlineare Aktivierungsfunktion würde sich das neuronale Netz wie ein einschichtiges Perzeptron verhalten, da die Kombination linearer Funktionen unabhängig von der Anzahl der Schichten immer nur eine lineare Funktion darstellt [11]. Es gibt verschiedene Arten von Aktivierungsfunktionen (siehe Abbildung 2.2).

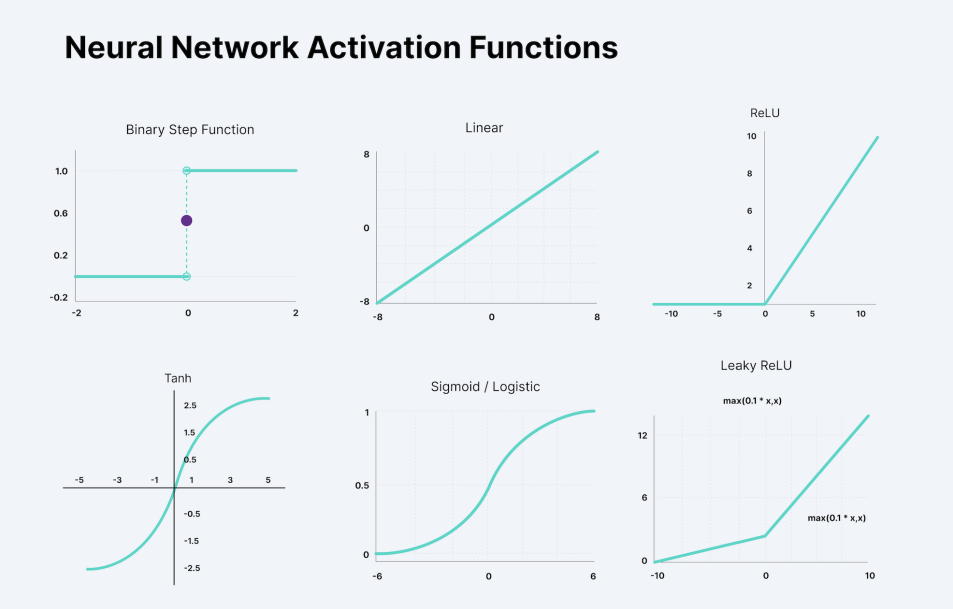


Abbildung 2.2: Unterschiedliche Aktivierungsfunktionen definieren unterschiedliche Wertebereich für die Ausgabe neuronaler Netze [12].

2.3 Kostenfunktionen

In der Regel wird bei neuronalen Netzen versucht, den Fehler (die Differenz zwischen der gewünschten Ausgabe und der Prädiktion (tatsächlichen Ausgabe des neuronalen Netzes) zu minimieren. Daher wird die Zielfunktion oft als Kostenfunktion oder Verlustfunktion bezeichnet und der durch die Verlustfunktion (Eng. Loss function) berechnete Wert wird einfach als **"Verlust"** (Eng. **Loss**) bezeichnet.

Die Wahl der Kostenfunktion steht in direktem Zusammenhang mit der Aktivierungsfunktion, die in der Ausgangsschicht des neuronalen Netzes verwendet wird. Diese beiden Entwurfselemente sind miteinander verbunden [13].
Es gib zahlreiche Kostenfunktionen wie *Maximum Likelihood*, *Mean Squared Error Loss*, *Cross-Entropy Loss (or Log Loss)*, *Wasserstein Loss* und ... .

In dieser Arbeite liegt die Konzentration auf **Cross-Entropy Loss (or Log Loss)** and **Wasserstein Loss** (*mehr dazu in Kapitel 4*).

2.4 Backpropagation

Backpropagation ist eine Methode, um die Parameter (Gewichte und Biases) des neuronalen Netzes in die richtige Richtung zu verändern. Zunächst wird der Loss-Term berechnet, und dann werden die Parameter des neuronalen Netzes in umgekehrter Reihenfolge mit einem Optimierungsalgorithmus (z. B. Gradient Decent, Stochastic gradient descent (SGD), Adam usw.) unter Berücksichtigung dieses berechneten Loss angepasst [14]. Mit anderen Worten: Backpropagation zielt darauf ab, durch Anpassen der Gewichte und Biases des Netzes die Kostenfunktion zu minimieren. Der Anpassungsgrad wird durch die Gradienten der Kostenfunktion in Abhängigkeit von den Parametern bestimmt.

2.5 Der Lernprozess eines neuronalen Netzes

Das Bemerkenswerte an neuronalen Netzen ist ihre Fähigkeit, aus den Daten zu lernen. Sie müssen nicht programmiert werden, sondern lernen das Programm von selbst [11]. Mit anderen Worten, die neuronalen Netze lernen während des Trainings eine Funktion f, so dass $f(x)$ auf y abbildet, oder vereinfacht gesagt, sie lernen, wie sie x (d. h. Merkmale oder, traditioneller, unabhängige Variable(n)) nehmen können, um y (das Ziel, die Antwort oder, traditioneller, die abhängige Variable) zu prognostizieren [15].

Die grundlegenden Schritte des Trainingsprozesses sind die folgenden [16]:

1. **Initialisieren** des Netzes mit zufälligen Gewichten.
2. **Forwardpropagation**:
    - Weitergabe der Eingabedaten durch das Netz, um ihre Prädiktion zu erhalten.
    - Berechnung des Verlusts (loss) durch eine Kostenfunktion.
3. **Backpropagation**:
    - Die Loss-information wird schichtweise auf alle Parameter übertragen, und zwar von der Ausgangsschicht zur Eingangsschicht. Die Neuronen der versteckten Schicht erhalten jedoch entsprechend ihrem relativen Beitrag zur ursprünglichen Ausgabe nur einen Bruchteil des Gesamtsignals des Losses.
    - die Aktualisierung der Parameter mit einer Optimierungsfunktion (z.b. Gradient decent oder SGD) in einer Weise, dass der Gesamtfehler reduziert wird.
4. **Fortsetzen der Iteration** in den vorherigen Schritten, solange bis ein gutes Modell entsteht.

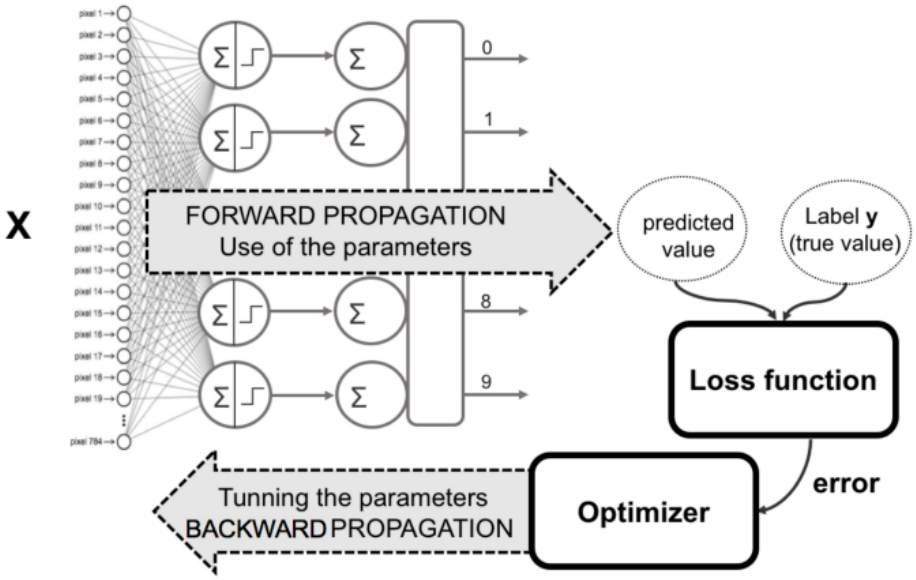


Abbildung 2.3: Der Lernprozess eines neuronalen Netzes [14]

2.6 Überwachtes und unüberwachtes Lernen

Die **Machine-Learning-Algorithmen** lassen sich nach dem Lernprozess in *überwachtes*, *unüberwachtes* und *verstärkendes* Lernen kategorisieren [17].

Überwachtes Lernen

Es wird anhand der Daten x mit Label y ein Modell trainiert, das mit Hilfe von y eine Funktion f lernt, um x zu repräsentieren (z.B. Klassifikation, RegressionObjekt Erkennung, Semantik Segmentation, usw.).

Unüberwachtes Lernen

Es werden nur die Daten x ohne Label verwendet, um ein Modell zu trainieren, das die verborgene oder grundlegende Struktur dieser Daten x lernt, um die neuen Daten ähnlich wie x zu erzeugen (z.B. Clustering, Autoencoders, GANs usw.).

Verstärkendes Lernen

Es befasst sich mit dem Problem eines Agenten, der lernt, in einer dynamischen Umgebung zu handeln, indem er die beste Folge von Aktionen findet, die eine Belohnungsfunktion maximiert. Die Grundidee ist, dass der Agent die interaktive Umgebung erkundet. Je nach den Beobachtungserfahrungen, die er macht, ändert er seine Handlungen, um höhere Belohnungen zu erhalten.

3 Generative Modelle

In diesem Abschnitt wird zunächst ein kurzer Überblick über die generative Modellierung und deren unterschied zur diskriminativen Modellierung gegeben. Danach wird es GANs näher betrachtet und es wird versucht einen besseren Überblick über die Komponenten und Strukturen dieser Modelle zu gewinnen und ein grundlegendes Verständnis hinsichtlich ihrer Funktionsweise zu erlangen.

3.1 Generative Modellierung

Die generative Modellierung ist eine unüberwachte Lernmethode beim maschinellen Lernen. Sie umfasst das automatische Erkennen und Erlernen von Regelmäßigkeiten oder Mustern in den Eingabedaten. Diese Methode kann in einem Modell verwendet werden, um ein neues Muster zu erzeugen, das dem ursprünglichen Datensatz ähnlich ist [18].

Unterschied zwischen generativen und diskriminativen Modellen [19]
Um zu verstehen, was die generative Modellierung bezweckt und warum dies wichtig ist, ist es sinnvoll, sie mit ihrem Pendant, der diskriminativen Modellierung, zu vergleichen.
Bei der diskriminativen Modellierung handelt es sich um eine überwachte Lernmethode, bei der das Modell eine Funktion erlernt, die eine Eingabe auf eine Ausgabe abbildet, wobei ein gelabelter Datensatz verwendet wird. Das ultimative Ziel von diskriminativen Modellen liegt darin, eine Klasse (z.B. Katze) von einer anderen Klasse (z.B. Hund) zu trennen ($Klasse(Katze|Hund)$).
Generative Modelle konzentrieren sich auf die Verteilung der einzelnen Klassen in einem Datensatz, und die Lern-Algorithmus neigen dazu, die zugrunde liegenden Patterns oder die Verteilung der Datenpunkte zu modellieren. Außerdem enthält das Modell ein stochastisches (zufälliges) Element, das die einzelnen vom Modell erzeugten Samples beeinflusst.

Der mathematische Unterschied zwischen generativer und diskriminativer Modellierung:

- **Die diskriminative** Modellierung ermittelt die bedingte Wahrscheinlichkeit $p(y|x)$.
  - die Wahrscheinlichkeit, dass die Eigenschaften (z. B. feuchte Nase, herausgestreckte Zunge, Schnurren usw.) x zur Klasse y (z.B. Hund) gehört.
- **Die generative** Modellierung ermittelt die kombinierte Wahrscheinlichkeit (Eng. joint Probability) $p(x, y)$, oder nur $p(x)$, wenn es keine Label gibt.
  - die Wahrscheinlichkeit, dass die Klasse y (Hund) die Eigenschaften x (z. B. feuchte Nase, herausgestreckte Zunge, Schnurren uzw.) aufweist.

Es gibt Zahlreiche Beispile für das generatives Model. **Variational Autoencoder (VAE)** und **Generative Adversarial Network (GANs)** sind zwei moderne Beispiele für ein generatives Modell. Der Schwerpunkt dieser Arbeit wird hauptsächlich auf Generative Adversarial Networks (GANs) gelegt.

3.2 Generative Adversarial Networks (GANs)

Generative Adversarial Networks (GANs) sind ein Beispiel für generative Modelle. Dieses Framework und seine erste empirische Demonstration wurde in dem Artikel von Ian Goodfellow, et al. aus dem Jahr 2014 mit dem Titel „Generative Adversarial Networks“ veröffentlicht [20]. Dieses Netzwerk enthält zwei Modelle, die gegensätzlich trainiert werden. Es basiert auf einem spieltheoretischen Ansatz mit dem Ziel, ein Nash-Gleichgewicht (Eng. Nash Equilibrium) zwischen den beiden Netzen, **Generator** und **Discriminator**, zu finden. Das Generator-Modell **G** lernt, die Datenverteilung zu erfassen, und das Discriminator-Modell **D** schätzt die Wahrscheinlichkeit, dass eine Probe aus der Trainingsdatenverteilung stammt und nicht aus den vom Generator-Modell generierten Daten. Grundsätzlich besteht die Aufgabe des Generators darin, natürlich aussehende Bilder zu erzeugen, und die Aufgabe des Discriminators ist es, zu entscheiden, ob das Bild gefälscht oder echt ist. Man kann sich dies als ein **Minimax-Spiel**[1] (Eng. minimax game) für zwei Spieler vorstellen, bei dem sich die Leistung beider Netzwerke mit der Zeit verbessert. In diesem Spiel versucht der Generator, den Discriminator zu täuschen, indem er so weit wie möglich echte Bilder erzeugt (*Maximierung der Wahrscheinlichkeit, dass D einen Fehler macht*), und der Discriminator versucht, nicht vom Generator getäuscht zu werden, indem er seine Unterscheidungsfähigkeit verbessert (*Minimierung der Wahrscheinlichkeit, dass D einen Fehler macht*). Die nachstehende Abbildung zeigt die grundlegende Architektur des GAN.

[1] Der Begriff ”Minimax” ist eine Entscheidungsregel in der Spieltheorie. Dies bezieht sich auf die Minimierung des Verlustes, der entsteht, wenn der Gegner die Strategie wählt, die den maximalen Verlust ergibt. Das Minimax kann später ein Nash-Gleichgewicht zur Folge haben [21].

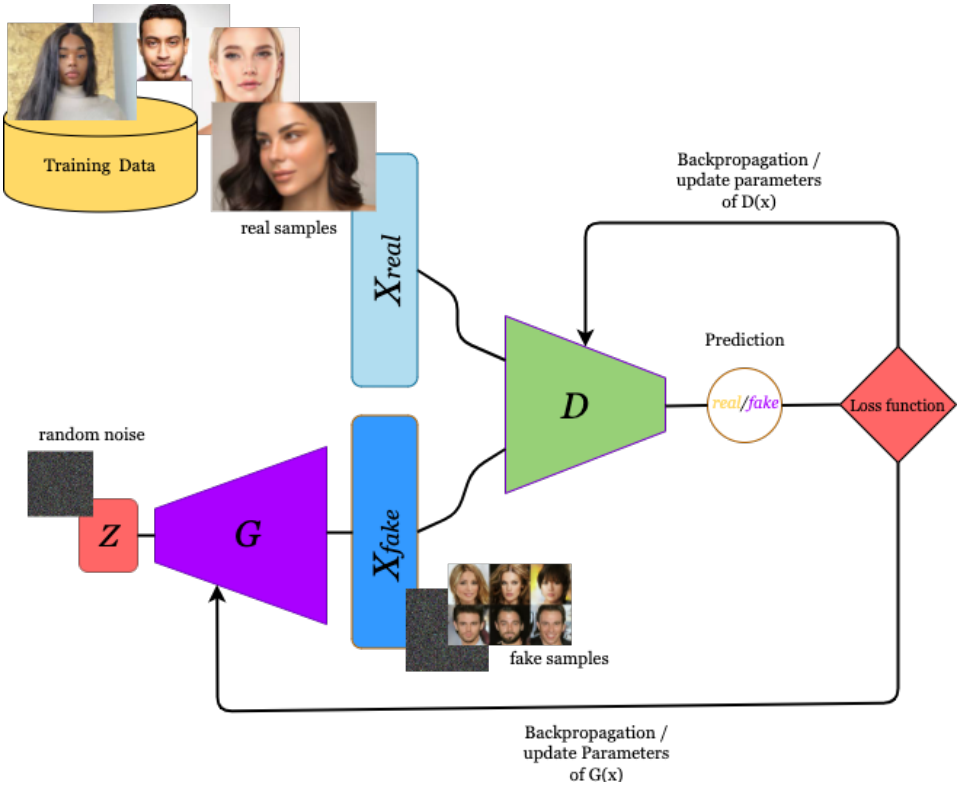


Abbildung 3.1: Generative Adversarial Networks (GANs) nach [22]

3.2.1 Discriminator

Das **Discriminator-Netzwerk D** ist ein **Classifier**, der lernt, die Eingabedaten in zwei Klassen zu unterteilen *(echt und gefälscht)* [22]. Das Modell[2] nimmt eine Eingabe x mit den Parametern θ_d und erzeugt einen einzelnen Skalar als Ausgabe *(1=real oder 0=fake)*. Das Discriminator-Modell $D(x)$ hat außerdem eine Verlustfunktion, die den Fehler des Modells auf der Basis seiner Prognose berechnet und diese zur Optimierung der Gewichte durch Backpropagation verwendet (siehe Abbildung 3.1) [23].

3.2.2 Generator

Das **Generator-Modell G** nimmt einen zufälligen Gaußschen Rauschvektor z als Eingabe und verwendet die Parameter θ_g, um eine Ausgabe x *(gefälschte Daten)* zu erzeugen. Die Dimensionalität des Rauschvektors z muss nicht unbedingt mit der Dimensionalität von x übereinstimmen [22]. Bei diesem Modell hat $G(z)$ keine eigene Verlustfunktion, sondern verwendet die Verlustfunktion des Discriminator, um ein Feedback für seine gefälschten Samples zu gewinnen und seine Parameter durch Backpropagation zu aktualisieren (siehe Abbildung 3.1) [23]. Die folgende Tabelle gibt einen Überblick über die beiden Modelle.

[2]Es ist üblicherweise ein mehrschichtiges Perzeptron.

	Discriminator	Generator
Eingabe	Ein echtes Bild aus Trainingsdaten oder ein gefälschtes Bild aus dem Generator	Ein zufälliger Rauschvektor z
Ausgabe	Eine Prognose für die Eingabe (real oder fake)	Ein gefälschtes Bild
Loss	Klassifizierung echter Bilder in die Kategorie *Echt* und gefälschter Bilder in die Kategorie *Gefälscht*.	Klassifizierung der gefälschten Bilder in die Kategorie Echt (vom Discriminator)

Tabelle 3.1: Die Eingabe und Ausgabe des Discriminators und des Generators [22].

3.2.3 Trainieren von GANs-Modellen (Minimax-Spiel)

Beim Trainieren des GAN-Frameworks treten der **Generator G** und der **Discriminator D** als zwei Konkurrenten in einem Spiel gegeneinander an. Jeder Spieler wird durch eine differenzierbare Funktion ($D(x)$ und $G(z)$) dargestellt, die durch eine Reihe von Parametern gesteuert wird.

Das Spiel lässt sich in zwei Szenarien abspielen. In einem Szenario werden Trainingsbeispiele x zufällig aus dem Datensatz entnommen und als Eingabe für den ersten Spieler, den Discriminator $D(x)$, verwendet. Hier sind x die echten Daten und das Ziel des Discriminators ist es, die Wahrscheinlichkeit auszugeben, dass seine Eingabe echt und nicht gefälscht ist. Das heißt, Ziel ist es, $D(x)$ nahe 1 zu bringen.

Im zweiten Szenario, an dem beide Spieler beteiligt sind, nimmt der Generator G die Eingabe z (zufällige Samples aus der Prior-Verteilung $P_z(z)$), $G(z)$ und erzeugt gefälschte Samples x, um sie an den Discriminator $D(G(z))$ zu übergeben. Der Discriminator ist bemüht, $D(G(z))$ gegen 0 zu bringen (gefälschte Samples), während der Generator versucht, dieselbe Quantität gegen 1 zu bringen (echte Samples), um den Discriminator zu täuschen. Hier wird D trainiert, um die Wahrscheinlichkeit zu *maximieren*, sowohl den echten als auch den gefälschten Samples das richtige Label zuzuweisen. Dabei wird G trainiert, $log(1 - D(G(z))$ zu *minimieren*. D und G spielen ein Minimax-Spiel mit der folgenden Wertfunktion nach Goodfellow u.a. [1].

$$\min_G \max_D V(D,G) = \mathbb{E}_{x \sim P_{data}(x)}[logD(x)] + \mathbb{E}_{z \sim P_z(z)}[log(1 - D(G(z))] \tag{3.1}$$

- $V(D,G)$:*Wert-Funktion (Eng. value function)*
- $\mathbb{E}$: *der Erwartungswert*
- $D(x)$: *die Wahrscheinlichkeit, dass x aus den realen Daten stammt und nicht aus P_g (generierte Samples)*
- $D(G(z))$: *die Wahrscheinlichkeit, dass $G(z)$ aus dem gefälschten oder erzeugten Rauschen z stammt*

Die Gleichung 3 besteht aus zwei Termen. Der erste Term stellt die Prädiktion von D für die realen Daten dar.

$$\mathbb{E}_{x \sim P_{data}(x)}[logD(x)] \tag{3.2}$$

Dieser Term ist die Erwartungswert für die Ausgabe des Logarithmus von D, wenn die Eingabe aus der Verteilung der realen Daten stammt, oder anders ausgedrückt, es ist der Durchschnittswert der Prädiktionen von D, wenn ein Bündel von realen Daten abgetastet und an D gegeben wird. Hier ist G überhaupt nicht beteiligt.

Der zweite Term stellt die Prädiktion von D für die gefälschten Daten dar.

$$\mathbb{E}_{z \sim P_z(z)}[log(1 - D(G(z))] \tag{3.3}$$

Es handelt sich um einen Erwartungswert für die Ausgabe von log 1 minus der Prognose des D für die gefälschten Proben. Der Erwartungswert ist einfach der Durchschnittswert der Prädiktion des D, wenn D eine Reihe von gefälschten Samples, die von G erzeugt wurden, als Eingabe erhält.

Im Allgemeinen beabsichtigt D, diese beiden Terme zu maximieren und G, sie zu minimieren. D maximiert diese Terme, indem es die echten Samples als echt ($D(x)$ nahezu 1) und die falschen Samples als falsch ($G(z)$ nahezu 0) vorhersagt ($1 + 1 - G(z) = 1 + 1 - 0.3 = 1.7$). Andererseits minimiert G diese Terme, indem es gute Samples erzeugt, die von D als echt klassifiziert werden können und $G(z)$ auf 1 annähert ($1 - G(z) = 1 - 0.8 = 0.2$)

Nach mehreren Trainingsschritten entsteht, wenn beide Modelle ausreichend Kapazität haben, ein Nash-Gleichgewicht (Eng. Nash Equilibrium) in diesem Spiel, wobei D und G einen Punkt erreichen, an dem sie sich nicht mehr verbessern können und $D(x) = \frac{1}{2}$ für alle x.

Abbildung 3.2 veranschaulicht den Trainingsprozess von GANs im 2D-Raum.

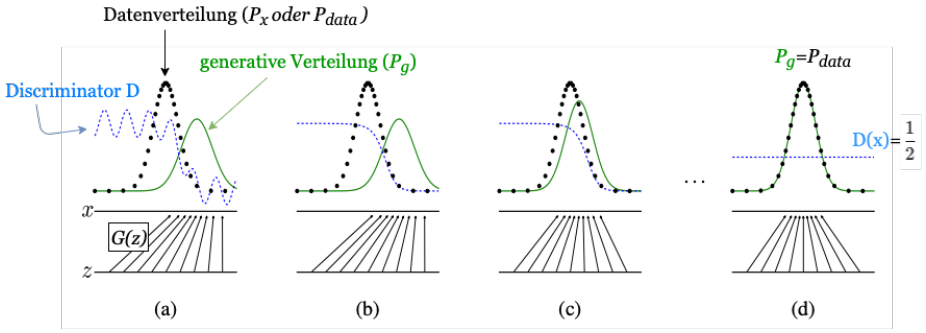


Abbildung 3.2: Eine einfache Darstellung, wie ein GAN-Modell trainiert wird. Hier sind die untere und die obere horizontale Linie jeweils der Wertebereich für z *(Gleichverteilung)* und x, und die schwarzen Pfeile zeigen, wie $G(z)$, z auf x in einer *ungleichmäßigen (non-uniform)* Weise abbildet, so dass P_g (*grün*) an den Stellen größer ist, an denen die z-Werte dichter zusammengeführt werden. Das Ziel beim Training des GAN ist es, die generative Verteilung P_g (*grüne Linie*) an die Datenverteilung P_{data} (*schwarze Linie*) anzugleichen ($P_g = P_{data}$ *und* $D(x) = \frac{1}{2}$, *siehe d*). Dies geschieht durch gleichzeitige Aktualisierung der Verteilung von D (*blaue, gestrichelte Linie*) und G (*grüne Linie*). Die Aktualisierung von D führt dazu, dass das D-Netz seine Kompetenz zur Unterscheidung zwischen echten und unechten Samples verbessert. Andererseits leitet der Gradient von D bei der Aktualisierung von G das $G(z)$ in Regionen (*von a bis d*), in denen es mit größerer Wahrscheinlichkeit als echte Daten klassifiziert werden kann [1].

4 GAN-Modelle und Umsetzung

In diesem Kapitel werden hauptsächlich das Konzept und die Umsetzung von drei ausgewählten Modellen von GANs, nämlich Vanilla oder Standard GAN (Ian Goodfellow, et al. 2014), Deep Convolutional GAN (DCGAN), Wasserstein GAN with Gradient Penalty (WGAN-GP), die in dieser Arbeit verwendet werden detailliert erläutert.

4.1 Standard GAN (SGAN)

Dieses Modell ist das allererste und die grundlegende Struktur von GANs, die von Ian Goodfellow und seinen Kollegen im Jahr 2014 vorgestellt wurde. Dieses Modell ist ein mehrschichtiges Perzeptron und verwendet lineare Schichten in seiner Architektur. Für das Training dieses Modells wird **Binary Cross Entropy (BCE) Loss** verwendet. Außerdem benutzt es **Batch Normalization** (Batch-Norm), um die Eingaben zu normalisieren und die eingesetzten Aktivierungsfunktionen sind ReLU, LeakyReLU und Sigmoid.

4.1.1 Binary Cross Entropy (BCE) Loss

Die Binary Cross Entropy-Funktion (BCE) wird für das Training von GANs verwendet. Sie ist für diese Modelle nützlich, da sie sich speziell für die Klassifizierungsaufgaben mit zwei Kategorien (Binary Classification) wie real (1) und fake (0) eignet. Die folgende Gleichung stellt die BCE-Loss-Funktion dar [24].

$$J_{bce} = -\frac{1}{m}\sum_{i=1}^{m}\Big[y_i \; log\;(p_i) + (1-y_i)\; log\;\big(1-p_i\big)\Big] \tag{4.1}$$

wobei:

m *Anzahl der Trainings-Proben (in einem Mini-Batch)*

y_i *Label für das Trainings-Probe i*

p_i *Modellprognose für Probe i*

Da es zwei Klassen ($class_{real}$ und $class_{fake}$) gibt, die durch Discriminator klassifiziert werden, gibt es auch zwei Labels. Normalerweise wird das Label für echte Proben als eins ($y_{i=real} = 1$) und das Label für gefälschte Proben als null ($y_{i=fake} = 0$) bezeichnet.

Der Ausdruck $\big(y_i \;\; log(p_i)\big)$ gibt die Prognose für die realen Proben und der Ausdruck

$\big((1 - y_i)\ log(1 - p_i)\big)$ die Prognose für die gefälschten Proben, denn wenn man das reale Label ($y = 1$) in die Gleichung einfügt, wird der zweite Ausdruck *Null* und bei den gefälschten Labeln ($y = 0$) ist der erste Ausdruck *Null*.

Der Ausdruck $\big(-\frac{1}{m}\sum_{i=1}^{m}\big)$ errechnet den Durchschnitt der Verlustwerte in einem Mini-Batch und gibt einen positiven Wert zurück.

BCE-Loss wird durch die `torch.nn.BCELoss` Klasse in Pytorch für Standard-GAN und DCGAN implementiert.

4.1.2 Batch Normalization (Batch-Norm)

Eine Schwierigkeit beim Training eines Deep-Learning-Modells wie GANs besteht darin, dass sich bei der Backpropagation und der Aktualisierung der Gewichte des Modells nach jedem Mini-Batch die Verteilung der Eingaben in die Netzschichten ändern kann. Dieses Phänomen wird als *Covariate-Shift* bezeichnet [25]. Außerdem ist es problematisch, wenn die Gewichte des Netzes nicht in einem vernünftigen Wertebereich verbleiben. Denn zu große Werte im Backward Pass einen so genannten „*Exploding-Gradient*“ und zu kleine Werte einen so genannten „*Vanishing-Gradient*“ verursachen.

Um ein solches Problem zu vermeiden, wurde in der vorliegenden Arbeit die Technik **Batch Normalization** (Batch-Norm) verwendet [26]. Die Batch-Norm normalisiert die Eingaben in eine Schicht bei jedem Mini-Batch und stabilisiert damit den Lernprozess und reduziert drastisch die Anzahl der Training-Epochen, die für das Training von Deep Networks erforderlich sind [27].

Eine Batch-Norm berechnet den Mittelwert und die Standardabweichung der Eingabewerte in jedem Mini-Batch. Sie normalisiert die Eingabewerte, indem sie sie vom Mittelwert subtrahiert und durch die Standardabweichung dividiert. Außerdem gibt es zwei gelernte Parameter, die den normalisierten Wert skalieren (Gamma) und verschieben (Beta). Die Ausgabe ist einfach die normalisierte Eingabe, skaliert mit gamma und verschoben mit beta (vgl. Alg.[1] 2).

[1] Das Epsilon (ϵ) in Alg. 2 verhindert, dass der Nenner negativ wird.

Algorithm 1 Batch Normalization nach [26]

Input: Values of x over mini-batch $\mathcal{B} = \{x_{1...m}\}$;
Learned Parameters: γ, β
Output: $\{y_i = \mathbf{BN}_{\gamma,\beta}(x_i)\}$

1: $\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$ ▷ mini-batch mean

2: $\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$ ▷ mini-batch variance

3: $\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$ ▷ normalize

4: $y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathbf{BN}_{\gamma,\beta}(x_i)$ ▷ scale and shift

Batch-Norm-Schichten können hinter so genannte *„Dense"*- oder *„Convolutional"-Schichten* eingefügt werden, um die Ausgabe von diesen Schichten zu normalisieren.
Die Batch-Norm wird sowohl im Generator als auch im Discriminator umgesetzt. Dies wird in Pytroch durch die Schicht `torch.nn.BatchNorm1d` (für 2D- oder 3D-Eingaben) oder `torch.nn.BatchNorm2d` (für 4D-Eingaben) implementiert.

4.1.3 Training-Algorithmus von SGAN

Das Trainieren des WGAN-Modells erfolgt nach folgendem Algorithmus.

Algorithm 2 Training von Standard-GAN nach (Ian Goodfellow, et al. 2014) [1]

1: **for** number of training iterations **do**
2: **for** k steps **do**
3: • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, ..., \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
4: • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{data}(\boldsymbol{x})$.
5: • Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta d}\frac{1}{m}\sum_{i=1}^{m}\Big[logD\big(\boldsymbol{x}^{(i)}\big) + log\big(1 - D\big(G\big(\boldsymbol{z}^{(i)}\big)\big)\big)\Big].$$

6: **end for**
7: • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, ..., \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
8: • Update the generator by descending its stochastic gradient:

$$\nabla_{\theta g}\frac{1}{m}\sum_{i=1}^{m} log\big(1 - D\big(G\big(\boldsymbol{z}^{(i)}\big)\big)\big).$$

9: **end for**

4.2 Deep Convolutional GANs (DCGAN)

Deep Convolutional GANs (DCGAN) ist ein weiteres Netzwerk-Design für GAN. DCGAN wurde durch Radford und seine Kollegen im Jahr 2016 vorgestellt [28]. Dieses Modell besteht hauptsächlich aus sogenannten **Convolutional Layers** (Conv-Layers) ohne Max-Pooling und Fully Connected Layers. Dieses GAN-Modell verwendet *Strided Convolution* und *Transposed Convolution* für das *Downsampling* und *Upsampling*. Das Modell verwendet außerdem **Batch Normalization** (Batch-Norm) nach den Conv-Layers und die eingesetzten Aktivierungsfunktionen sind *ReLU*, *LeakyReLU* und *tanh*.

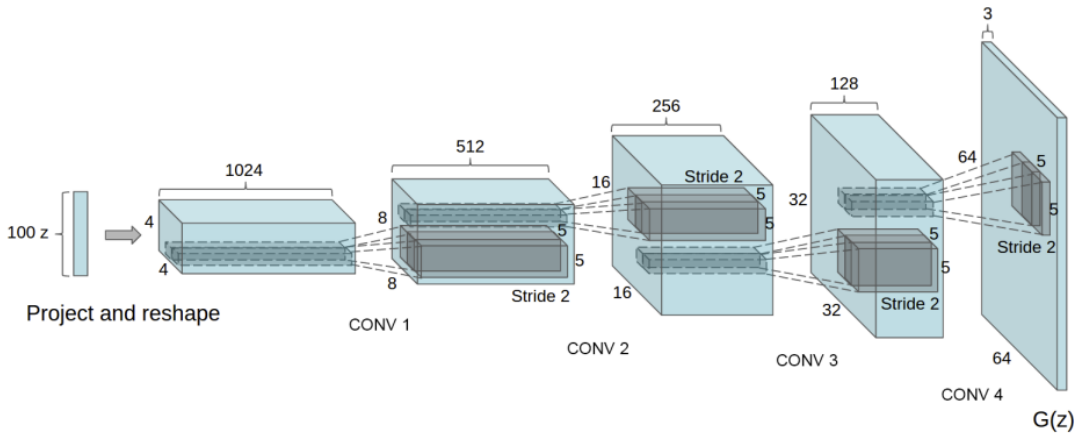


Abbildung 4.1: DCGAN-Netzentwurf für den Generator [28]

4.2.1 Convolutional Layers

Convolutions sind ein zentraler Bestandteil der Bildverarbeitung, weshalb sie in vielen GAN-Architekturen eine besonders wichtige Rolle spielen. Dies ist eine lineare Operation, bei der eine Menge von Gewichten (Filter oder Kernel) mit dem Input (Pixelwerte eines Bildes) multipliziert und deren Ergebnis summiert wird [13]. Die Convolutional Layers nutzen diese Operation, um die entscheidenden Merkmale wie Kanten in verschiedenen Bereichen eines Bildes zu erkennen.

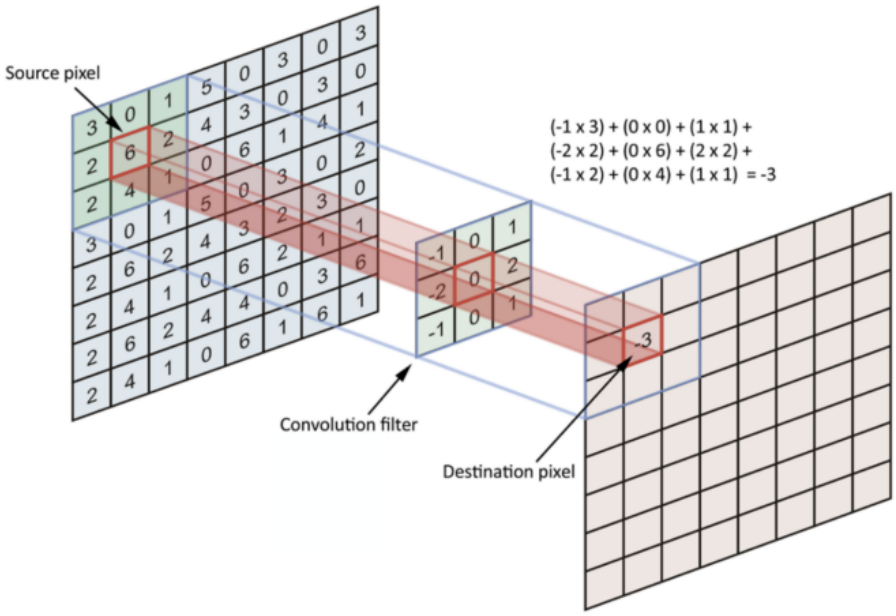


Abbildung 4.2: The convolution operation [29]

4.2.2 Filter (Kernel)

Der Filter oder Kernel ist eine Matrix aus realen Werten, und diese Werte werden beim Training gelernt. Während des Trainings werden verschiedene Filter erlernt, um verschiedene Merkmale aus dem Input-Bild zu extrahieren, z. B. Filter zur Erkennung von Merkmalen auf niedriger Ebene wie Linien und Kanten oder Filter zur Erkennung von Merkmalen auf hoher Ebene wie Augen, Nase oder Gesicht.

4.2.3 Stride

Stride modifiziert die Schrittgröße der Bewegung des Filters über den Input. Zum Beispiel, wenn $stride = 1$ ist, bewegt sich der Filter jeweils um ein Pixel oder eine Einheit. Die Erhöhung der Stride reduziert die Größe der Ausgabe (Feature-Map).

4.2.4 Padding

Bei der Convolution-Operation wird eine Anzahl von Pixeln als Rand um das Eingabebild hinzugefügt, um zu verhindern, dass die Informationen in den Ecken des Eingabebildes verloren gehen. Dies bewirkt, dass der Filter die Pixel in den Ecken wie die Pixel in der Mitte des Input-Bildes mehr als einmal abdeckt (vgl. Abbildung 4.3). Die Anzahl der Pixel, die zu einem Eingabebild hinzugefügt werden, wird als Padding bezeichnet. Das Padding kann das Ausgabebild größer als das Originalbild machen.

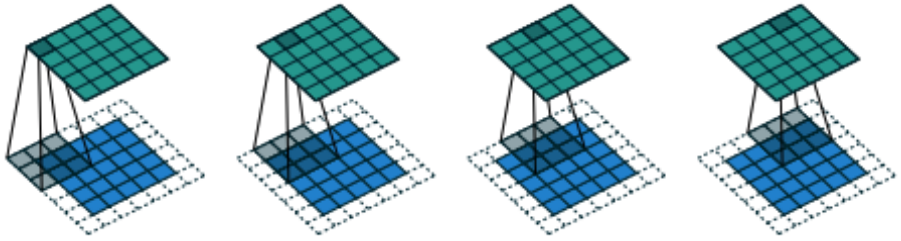


Abbildung 4.3: Ein 3 × 3 × 1-Filter (grau) wird über ein 5 × 5 × 1-Input-Bild (blau) bewegt, mit padding=1 und strides = 1, um den 5 × 5 × 1-Output (grün) zu erzeugen. [30].

4.2.5 Strided Convolution

Convolution-Operation mit Stride größer als eins (z.B. *stride*=2) werden als strided Convolution bezeichnet [31]. Dies ermöglicht es dem Netz, sein eigenes räumliches *Downsampling* zu erlernen (siehe Abbildung 4.4) [28].

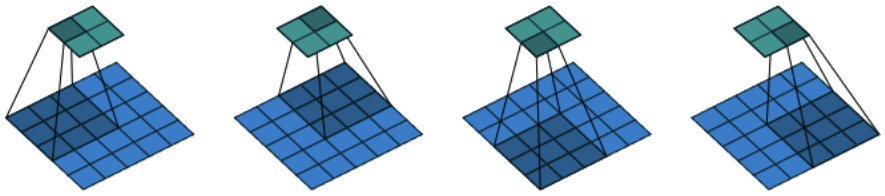


Abbildung 4.4: Ein 3 × 3 × 1-Filter (grau) wird über ein 5 × 5 × 1-Input-Bild (blau) bewegt, mit padding=0 und strides = 2, um den 2 × 2 × 1-Output (grün) zu erzeugen. [30].

Der Discriminator nutzt dieses Prinzip in seiner Convolutional-Layer, um die Größe der Eingabebilder für die Merkmalsextraktion[2] zu verringern (Downsampling). Um dies zu implementieren, wird die Klasse `torch.nn.Conv2d` in Pytorch verwendet.

[2] $f(x)$ entspricht dem durch den Discriminator extrahierten Feature-Vektor.

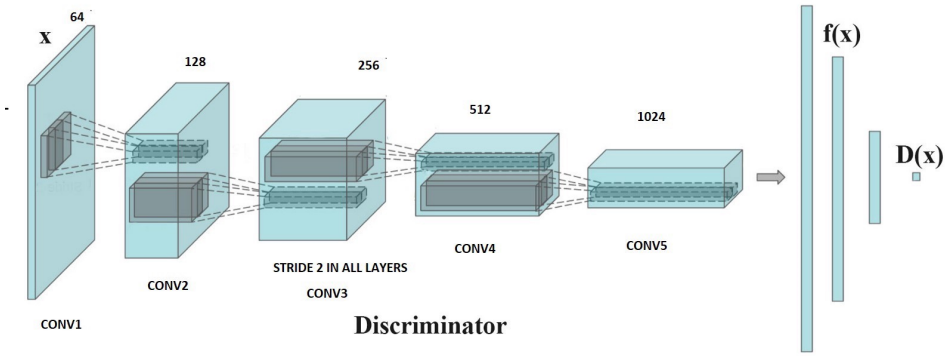


Abbildung 4.5: DCGAN-Netzentwurf für den Discriminator [32]

4.2.6 Transposed Convolution (Fractionally strided Convolutions)

Transposed Convolutions werden zum *Upsampling* der Eingabe auf eine gewünschte Ausgabe (Feature-Map) benutzt. Es handelt sich um eine Transformation in die entgegengesetzte Richtung von herkömmlichen Convolution. Transspose Convolution in Abbildung 4.6 ist äquivalent zu einer Convolution mit 3 × 3 Filter über eine 4 × 4 Eingabe mit Einheitsschritten.

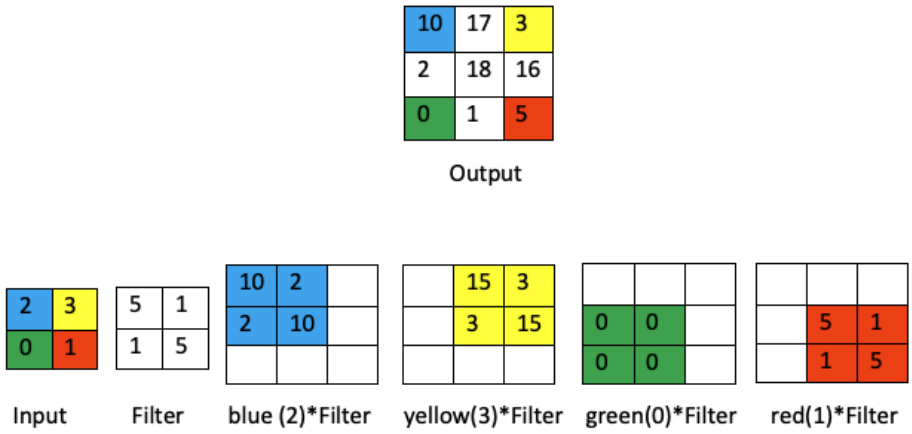


Abbildung 4.6: Ein 2 × 2 × 1-Filter wird über ein 2 × 2 × 1-Input-Bild bewegt, mit padding=0 und strides = 2, um den 2 × 2 × 1-Output (grün) zu erzeugen.

Die Convolutional Layer in Generator nutzt dieses Prinzip, um die Größe des Eingangsrauschens zu erhöhen und so z. B. gefälschte Bilder zu erzeugen, die so groß wie echte Bilder sind (Upsampling). Um dies zu implementieren, wird die Klasse `torch.nn. ConvTranspose2d` in Pytorch verwendet.

4.3 Wasserstein GAN with Gradient Penalty (WGAN-GP)

Wasserstein GAN with Gradient Penalty oder WGAN-GP, ist ein Generative Adversarial Network, das neben der Wasserstein-Loss-Formulierung auch eine Gradienten Norm Penalty verwendet, um Lipschitz Continuity zu erreichen [33]. In den folgenden Abschnitten werden die Konzepte, die das WGAN-GP verdeutlichen erläutert.

4.3.1 Wasserstein Distance oder Earth Mover's Distance (EMD)

Wasserstein Distance oder Earth Mover's Distance (EMD) ist der minimale Aufwand zur Angleichung einer Verteilung an eine andere. Dies ist sowohl von der Entfernung als auch von der Masse abhängig. Zum Beispiel gibt es in Abbildung 4.7 eine generierte und eine reale Verteilung. Die Earth Mover's Distance bestimmt, wie unterschiedlich diese beiden Verteilungen sind, indem sie den Aufwand für die Anpassung der generierten Verteilung an die reale Verteilung abschätzt oder anders ausgedrückt: Wenn die generierte Verteilung als ein Haufen Erde betrachtet wird, wie schwierig wäre es, diesen Haufen Erde zu bewegen, um ihn in die Form und Lage der realen Verteilung zu bringen?

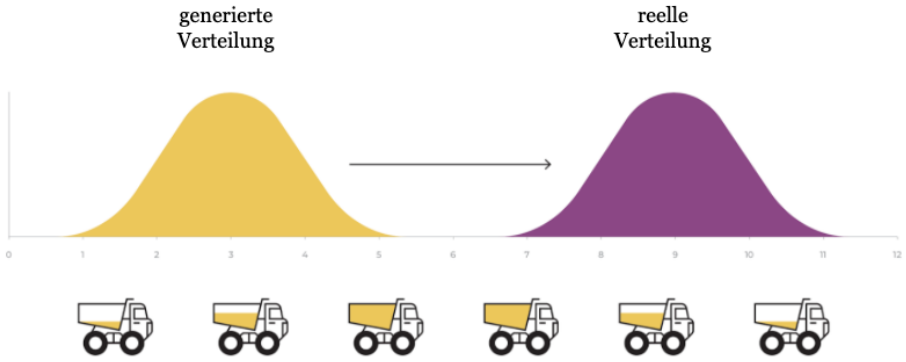


Abbildung 4.7: Wasserstein Distance (oder EMD) nach [34]

4.3.2 Wasserstein Loss (W-Loss)

Wasserstein Loss (W-Loss) nähert sich (approximiert) der EMD zwischen der realen und der generierten Verteilung an [35].

Die Funktion von W-Loss ist wie folgt dargestellt:

$$\min_{G} \max_{D} \; \mathbb{E}\left[D(x)\right] - \mathbb{E}\left[D(G(z))\right] \tag{4.2}$$

wobei:

$\mathbb{E}$ *der Erwartungswert*

$D(x)$ *die Wahrscheinlichkeit, dass x aus den realen Daten stammt*

$D(G(z))$ *die Wahrscheinlichkeit, dass* $G(z)$ *aus den gefälschten Daten stammt*

Der Discriminator in WGAN wird **Critic** genannt. Da er hier keine Klassifizierungsaufgabe übernimmt bzw. nicht zwischen *real* und *fake* diskriminiert. Der Critic versucht den Abstand zwischen der realen Verteilung und der gefälschten Verteilung zu maximieren, indem er die realen Bilder mit einem höheren Wert bewertet. Aber der Generator bemüht sich, die beiden Verteilungen so nah wie möglich zueinander zu schieben und die Differenz zu minimieren.

Die Ausgabe von W-Loss (Gleichung 4.2) kann eine beliebige Zahl $[-\infty\ +\infty]$ sein und ist nicht beschränkt. Dies erlaubt dem Critic, sehr große Zahlen auszugeben, die für neuronale Netze nicht effizient sind. Um den W-Loss für das Training von GANs effizient zu machen, ist eine spezielle Bedingung an die *Critic-Funktion* gestellt. Sie muss **1-Lipschitz-Continuous** (1-L-Continuous) sein.

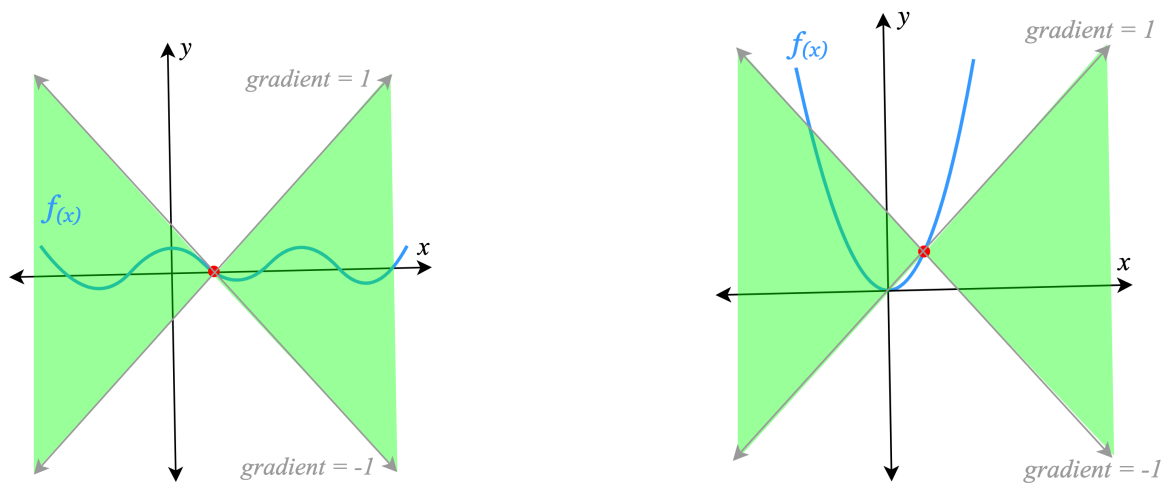
1-Lipschitz-Continuous: Eine Funktion wie die *Critics-Funktion* ist 1-L-Continuous, wenn die Norm ihres Gradienten *höchstens eins* ist. Das bedeutet, dass die Steigung (Gradient) an keinem Punkt größer als *eins* sein kann (vgl.Gleichung 4.3).

$$\|\nabla f(x)\|_2 \leq 1 \tag{4.3}$$

wobei:

∇ *der Gradient von der Funktion* $f(x)$

$f(x)$ *Funktion* f *mit einer Eingabe* x



(a) $f(x)$ ist 1-Lipschitz-Continuous.

(b) $f(x)$ ist nicht 1-Lipschitz-Continuous.

Abbildung 4.8: Der Gradient einer 1-L-Continuous Funktion muss überall höchstens 1 sein (*Der Funktionsgraph muss überall zwischen zwei grünen Teilen liegen*).

Neben **Weight-Clipping** ist **Gradient-Penalty** eine weitere Methode, um dem Critic *1-L-Continuity* aufzuerlegen.

4.3.3 Gradient-Penalty

Bei der **Weight-Clipping** werden die Gewichte des Critic gezwungen, Werte zwischen einem festen Intervall anzunehmen ($w = [-0.01, 0.01]$). Nachdem Sie die Gewichte während des Gradient-Descent aktualisiert haben, werden alle Gewichte außerhalb des gewünschten Intervalls abgeschnitten (Eng. clip). Das bedeutet im Wesentlichen, dass Gewichte, die über dem Intervall liegen, entweder zu hoch oder zu niedrig sind, auf den zulässigen Höchst- oder Mindestwert gesetzt werden.

Die **Gradienten-Penalty** ist eine weitere Möglichkeit, diese Bedingung (1-L-Continuity) erfüllen. Bei dieser Methode wird der W-Loss-Funktion ein **Regularisierungs-Term** hinzugefügt. Dieser Regularisierungs-Term *bestraft* den Critic, wenn seine Gradient-Norm größer als *eins* ist (vgl.Gleichung 4.3). Die Gleichung für den **Wasser Stein Loss with Gradient Penalty** (WGP-Loss) ist wie folgt:

$$\mathbf{L}_{gp} = \underbrace{\mathbb{E}\left[D(G(z))\right] - \mathbb{E}\left[D(x)\right]}_{W-Loss} + \underbrace{\lambda\, \mathbb{E}\left[\left(\|\nabla D(\hat{x})\|_2 - 1\right)^2\right]}_{Regularisierungs-Term\ oder\ Gradient-Penalty} \tag{4.4}$$

Im Regularisierungs-Term ist λ der Strafe-Koeffizient. Sein Wert bestimmt, wie stark dieser

Regularisierungs-Term gegenüber der W-Loss-Funktion gewichtet werden soll. Zusätzlich wird durch diesen Term die quadratische Differenz zwischen der Norm der Gradienten von Critic und 1 gemessen. Allerdings ist die Überprüfung dieser Gradienten an jedem möglichen Punkt des feature-space nahezu unmöglich oder zumindest nicht praktisch. Daher verwendet das WGAN-GP ein **interpoliertes Bild** ($\hat{x}$), das entlang gerader Linien zwischen Paaren von Punkten aus der realen Verteilung und der generierten Verteilung gesampelt wird (siehe Abbildung 4.9).

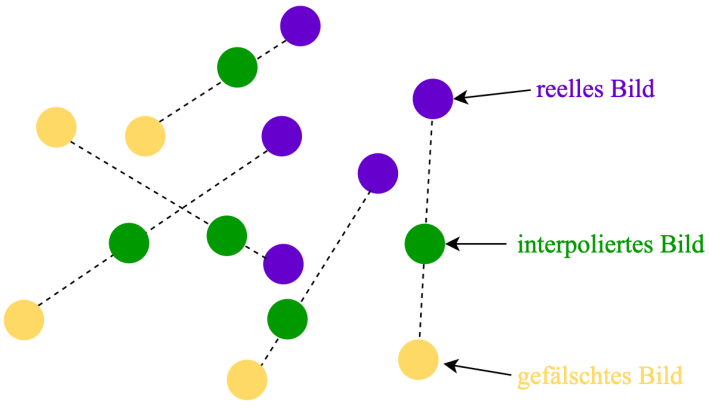


Abbildung 4.9: Bild-Interpolation zwischen der realen Verteilung und der gefälschten Verteilung im Feature Space

4.3.4 Training-Algorithmus von WGAN-GP

Das Trainieren des WGAN-Modells erfolgt nach folgendem Algorithmus.

Algorithm 3 Training von WGAN-GP nach [33]. Standard-Werte: $\lambda = 10$, $n_{critic} = 5$, $\alpha = 0.0001$, $\beta_1 = 0$, $\beta_2 = 0.9$

Require: The gradient penalty coefficient λ, the number of critic iterations per generator iteration n_{critic}, the batch size m, Adam hyperparameters α, β_1, β_2.
Require: initial critic parameters w_0, initial generator parameters θ_0.
while θ has not converged **do**
  for $t = 1, ..., n_{critic}$ **do**
    for $i = 1, ..., m$ **do**
      Sample real data $\boldsymbol{x} \sim \mathbb{P}_r$, latent variable $\boldsymbol{z} \sim p(\boldsymbol{z})$, a random number $\epsilon \sim \mathrm{U}[0,1]$.
      $\tilde{\boldsymbol{x}} \leftarrow G_\theta(\boldsymbol{z})$
      $\hat{\boldsymbol{x}} \leftarrow \epsilon \boldsymbol{x} + (1-\epsilon)\tilde{\boldsymbol{x}}$
      $L^{(i)} \leftarrow D_w(\tilde{\boldsymbol{x}}) - D_w(\boldsymbol{x}) + \lambda(\|\nabla_{\hat{\boldsymbol{x}}} D_w(\hat{\boldsymbol{x}})\|_2 - 1)^2$
    end for
    $w \leftarrow \mathrm{Adam}(\nabla_w \frac{1}{m} \sum_{i=1}^{m} L^{(i)}, w, \alpha, \beta_1, \beta_2)$
  end for
  Sample a batch of latent variables $\{z^{(i)}\}_{i=1}^{m} \sim p(\boldsymbol{z})$.
  $\theta \leftarrow \mathrm{Adam}(\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} -D_w(G_\theta(\boldsymbol{z})), \theta, \alpha, \beta_1, \beta_2)$
end while

4.4 Umsetzung

Für jedes Modell wurden folgende Hauptkomponenten implementiert:

- Laden von Daten und Datentransformation
- Discriminator Network
- Generator Network
- Training Loop
- Speichern und Laden von Modell-Check Points
- Evaluation
- Visualisierung
- Inference

4.4.1 Laden von Daten und Datentransformation

Die in dieser Arbeit implementierten GAN-Modelle werden auf dem Datensatz **CelebA** trainiert. Diese drei GAN-Modelle werden auf Bildern in der Größe `64x64` mit *RGB-Kanal* trainiert (`Input = 3x64x64`). Bevor die Trainingsdaten an diese Modelle weitergegeben werden, müssen sie auf die Größe `64x64` angepasst werden. Außerdem werden sie zentriert und in **Tensor** (eine Verallgemeinerung von Matrizen) umgewandelt. Die Tensor-Werte dieser Bilder werden mit `Mittelwert = 0,5` und `Standardabweichung = 0,5` normalisiert, um sie in den Bereich `[-1, 1]` zu bringen. Der `dataloader` übergibt die transformierten Bilder bei jedem Training an das Modell (für jede Trainings-Iteration so viele Bilder wie `batch_size = 128`). Um dies zu implementieren wird die Klasse `torchvision.transforms.Compose(transforms)` von *Pytorch* verwendet.

```
transform=transforms.Compose([
                        transforms.Resize([64, 64]),
                        transforms.CenterCrop(64),
                        transforms.ToTensor(),
                        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                        ])

path = './img_align_celeba'
dataset = datasets.ImageFolder(root=path, transform=transform)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

4.4.2 Discriminator Networks, Generator Networks und Training Loops

Diese GAN-Modelle basieren auf zwei verschiedenen Discriminator und Generator Architekturen. Discriminator und Generator in SGAN-Modell haben linear Layers (lineare Schichten), aber für DCGAN und WGAN-GP werden die sogenannten Convolutional Layers verwendet.

A: Discriminator Network von SGAN

Das Discriminator Network hat *vier* lineare Schichten. Nach jeder Schicht mit Ausnahme der letzten Schicht wird `LeakyReLU` als Aktivierungsfunktion verwendet, um das *„dying ReLU“* Problem und sein Resultat (*Nullgradient*) zu verhindern. Die Aktivierungsfunktion für die letzte Schicht ist `Sigmoid`, aber da sie in der BCE Loss integriert ist, wird sie hier nicht benötigt. Das Netzwerk nimmt die realen und gefälschten Bild-Tensoren mit der Größe von `3x64x64` (*12288 Dimensionen*) als Eingabe und die Ausgabegröße ist eine einzelne Zahl (*eindimensionaler Tensor*), die *Echt* oder *Gefälscht* darstellt.

```
class Discriminator(nn.Module):
    def __init__(self, img_dim=12288, hidden_dim=64):
        super(Discriminator, self).__init__()
        # Discriminator network layers
        self.disc = nn.Sequential(
            self.disc_layer(img_dim, hidden_dim * 4),
            self.disc_layer(hidden_dim * 4, hidden_dim * 2),
            self.disc_layer(hidden_dim *2, hidden_dim),
            self.disc_layer(hidden_dim, 1, final_layer=True)
        )

    # this method creat the component of generator networtks layers
    def disc_layer(self, input_dim, output_dim, final_layer=False):
        if not final_layer:
            return nn.Sequential(
                nn.Linear(input_dim, output_dim),
                nn.LeakyReLU(negative_slope=0.2, inplace=True)
            )
        else:
            return nn.Sequential(
            nn.Linear(input_dim, output_dim),
            )

    # it complets the forward pass:
    # get an image tensor and returns 1-dim tensor (fake/real)
    def forward(self, image):
        return self.disc(image)
```

Listing 4.1: Discriminator von SGAN

B: Generator Network von SGAN

Das Generator Network hat *fünf* lineare Schichten. Die Ausgaben jeder linearen Schicht mit Ausnahme der letzten Schicht werden an `BatchNorm1d` zur Normalisierung weitergegeben (siehe u.A. 4.1.2) und nach der Normalisierung an die *ReLU-Aktivierungsfunktion* gesendet. Die Aktivierungsfunktion für die letzte Schicht ist `Sigmoid`. Das Netzwerk nimmt einen Rauschvektor mit *64 Dimensionen* (`z_dim`) als Eingabe und die Ausgabe der letzten Schicht ist `3x64x64` (gefälschtes Bild mit `12288` Dimensionen mit der gleichen Größe wie der Discriminator-Eingang).

```
class Generator(nn.Module):
    #img_dim = 3*64*64 = 12288= size of output img
    def __init__(self, z_dim=64, img_dim=12288, hidden_dim=64):
        super(Generator, self).__init__()
        # generator network layers
        self.gen = nn.Sequential(
            self.gen_layer(z_dim, hidden_dim),
            self.gen_layer(hidden_dim, hidden_dim * 2),
            self.gen_layer(hidden_dim * 2, hidden_dim * 4),
            self.gen_layer(hidden_dim * 4, hidden_dim * 8),
            self.gen_layer(hidden_dim * 8, img_dim, final_layer=True)
        )

    # this method creat the component of generator networtks layers
    def gen_layer(self, input_dim, output_dim, final_layer=False):
        if not final_layer:
            return nn.Sequential(
                nn.Linear(input_dim, output_dim),
                nn.BatchNorm1d(output_dim),
                nn.ReLU(inplace=True)
            )
        else:
            return nn.Sequential(
                nn.Linear(input_dim, output_dim),
                nn.Sigmoid()
            )

    # it complets the forward pass:
    # get a noise tensorand returns generated images
```

```
30     def forward(self, nois):
31         return self.gen(nois)
```

Listing 4.2: Generator von SGAN

C: Discriminator Network von DCGAN und WGAN-GP

Das Discriminator Network im DCGAN und WGAN-GP Modell hat *fünf* Convolutional-Schichten (Strided Convolution). Die Ausgaben jeder Convolutional-Schicht mit Ausnahme der letzten Schicht werden an `BatchNorm2d` zur Normalisierung weitergegeben (siehe u.A. 4.1.2). Nach jeder `BatchNorm2d-Schicht` mit Ausnahme der letzten Schicht wird `LeakyReLU` als Aktivierungsfunktion verwendet. Das Netzwerk nimmt die realen und gefälschten Bild-Tensoren mit der Größe von `3x64x64` (*(Bildkanal oder Tiefe, Höhe, Breite)*) als Eingabe und nach einigen Convolution-Operation mit einem *Filter* von `4x4`, einem *Stride* von `2` (Strided Convolution, siehe u.A. 4.2.5) und einem *Padding* von `1` die (außer letzte Schicht) ist die Ausgabe ein *flacher Tensor* mit `512 Dimensionen, 1 Spalte und 1 Zeile` (Downsampling).

```
1  class Discriminator(nn.Module):
2     def __init__(self, img_chan=3, feature_maps=64):
3          super(Discriminator, self).__init__()
4          # discriminator network
5          self.discriminator = nn.Sequential(
6              # size = 3x64x64
7              self.creat_layer(img_chan, feature_maps,
8                               kernel=4, stride=2, padding=1),
9              # size = 64x32x32
10             self.creat_layer(feature_maps, feature_maps * 2,
11                              kernel=4, stride=2, padding=1),
12             # size = 128x16x16
13             self.creat_layer(feature_maps * 2, feature_maps * 4,
14                              kernel=4, stride=2, padding=1),
15             # size = 256x8x8
16             self.creat_layer(feature_maps * 4, feature_maps * 8,
17                              kernel=4, stride=2, padding=1),
18             # size = 512x4x4
19             self.creat_layer(feature_maps * 8, 1,
20                              kernel=4, stride=1, padding=0, final_layer=True),
21             # size = 512x1x1(flatten)
22         )
23
24     # creat disc-layer components
25     def creat_layer(self, in_channels, out_channels, kernel,
```

```
                            stride, padding, final_layer=False):
        if not final_layer:
            return nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel, stride, padding),
                nn.BatchNorm2d(out_channels),
                nn.LeakyReLU(0.2)
            )
        else:
            return nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel, stride, padding),
            )

    # it complets the forward pass:
    # get an image tensor and returns 512x1x1 tensor
    def forward(self, image):
        return self.discriminator(image)
```

Listing 4.3: Discriminator von DCGAN und WGAN-GP

D: Generator Network von DCGAN und WGAN-GP

Das Generator Network im DCGAN und WGAN-GP Modell besteht aus *fünf* Transposed Convolutional-Schichten (Upsampling). Die Ausgaben jeder Schichten mit Ausnahme der letzten Schicht werden an `BatchNorm2d` zur Normalisierung weitergegeben (siehe u.A. 4.1.2). Nach jeder `BatchNorm2d-Schicht` mit Ausnahme der letzten Schicht wird `LeakyReLU` als Aktivierungsfunktion verwendet. Die Aktivierungsfunktion für die letzte Schicht ist `Tanh`. Das Netzwerk nimmt einen Rauschvektor mit *100 Dimensionen* (`z_dim`) als Eingabe und die Ausgabe der letzten Schicht ist `3x64x64` (gefälschtes Bild mit Bildkanal oder Tiefe=3, Höhe = 64, Breite=64).

```
class Generator(nn.Module):
    def __init__(self, in_channels=3, z_dim=100, feature_maps=512):
        super(Generator, self).__init__()
        # generator network layers
        self.generator = nn.Sequential(
            # size = z_dim (100x1x1)
            self.create_layer(z_dim, 512, kernel=4, stride=1, padding=0),
            # size = 512x4x4
            self.create_layer(512,  256,  kernel=4, stride=2, padding=1),
            # size = 256x8x8
            self.create_layer(256,   128,  kernel=4, stride=2, padding=1),
            # size = 128x16x16
```

```
            self.create_layer(128,   64,  kernel=4, stride=2, padding=1),
            # size = 64x32x32
            self.create_layer(64,   3,    kernel=4, stride=2, padding=1,
    final_layer=True),
            # size = 3x64x64
        )

    def create_layer(self, in_channels, out_channels, kernel, stride,
                           padding, final_layer=False):
        if final_layer:
            return nn.Sequential(
                nn.ConvTranspose2d(in_channels, out_channels, kernel, stride,
                                   padding, bias=False),
                nn.Tanh())
        else:
            return nn.Sequential(
                nn.ConvTranspose2d(in_channels, out_channels, kernel, stride,
                                   padding, bias=False),
                nn.BatchNorm2d(out_channels),
                nn.ReLU(True))

    # it complets the forward pass:
    # get a noise tensorand returns generated images (3x64x64)
    def forward(self, noise):
        return self.generator(noise)
```

Listing 4.4: Generator von DGAN und WGAN-GP

E: Training Loop von SGAN und DCGAN

Das Trainingskonzept dieser drei GAN-Modelle ist nahezu identisch. Für das Training von SGAN und DCGAN wird *BCE* als Verlustfunktion verwendet und für die Aktualisierung der *Gewichte* und der *Learning rate* wird **Adam Optimizer** verwendet (siehe ALg.2).

Der Trainingsprozess wird wie folgt beschrieben:

- **Trainingsdauer**
  Die Modelle werden 50 Mal (`num_epoch=50`) mit allen Bildern (über 200 Tausend) des Datensatzes trainiert. Bei jedem dieser 50 Male wird ein Mini-Batchvon Bildern (`batch_size = 128` Bilder) durch (`dataloader` aus dem Datensatz geladen und an die Modelle weitergegeben, nicht alle Bilder auf einmal (siehe auch Algorithmus 2).

```
for epoch in range(n_epochs):
    for real, _ in tqdm(dataloader):
```

- **Training Discriminator Network:**
  1. Erzeugen eines Mini-Batchvon zufälligen Rauschvektoren mit 64 Dimensionen und Senden dieser als Eingabe an den Generator, um ein Mini-Batchvon gefälschten Bildern zu erzeugen (siehe Zeile 1-2 im Cod und vgl.Alg.2).
  2. Laden eines Batchs realer Bilder aus dem Datensatz (Zeile 4).
  3. Senden der echten und gefälschten Bilder einzeln an das Discriminator Network, um vorherzusagen, ob sie echt oder gefälscht sind (Zeile 8-9).
  4. Senden der Discriminator-Vorhersagen für echte Bilder mit dem Label *real (1)* und für gefälschte Bilder mit dem Label *fake (0)* an *BCE Loss* (siehe Gleichung 4.1), um die *Fehler* zu berechnen (Zeile 11-14).
  5. Der Discriminator-Fehler ist der Durchschnitt der Fehler von echten und gefälschten Bildern. Der durchschnittliche Fehler wird für den *Backward Pass* verwendet. Im letzten Schritt werden die Gewichte und Parameter von Discriminator Network aktualisiert (Zeile 16-19).

```
1  noise = torch.randn(cur_batch_size, z_dim, device=device)
2  fake = gen(noise)
3
4  real = real.view(cur_batch_size, -1).to(device)
5
6  # detach(): ruturn new Tesnor without gradient requirement
7  #(stop updating G in this step).
8  disc_fake_pred = disc(fake.detach())e cal
9  disc_real_pred = disc(real)
10
11 disc_fake_loss = criterion(disc_fake_pred,
12                             torch.zeros_like(disc_fake_pred))
13 disc_real_loss = criterion(disc_real_pred,
14                             torch.ones_like(disc_real_pred))
15
16 disc_avg_loss = (disc_fake_loss + disc_real_loss) / 2
17
18 disc_avg_loss.backward(retain_graph=True)
19 disc_optimizer.step()
```

- **Training Generator Network:**
  1. Erzeugen eines Mini-Batch von zufälliger Rauschvektoren mit 64 Dimensionen und Senden dieser als Eingabe an den Generator, um einen Mini-Batchgefälschter Bilder zu erzeugen (siehe Zeile 1-2 im Cod und vgl.Alg.2).
  2. Senden der gefälschten Bilder an das Discriminator Network, um vorherzusagen, ob sie echt oder gefälscht sind (Zeile 4).

3. Senden der Discriminator-Vorhersagen für gefälschte Bilder dieses Mal mit dem Label *real (1)* an *BCE Loss*, um die *Fehler* zu berechnen (Zeile 6).
4. Verwendung des Fehlers beim *Backward Pass* zur Aktualisierung der Gewichte und Parameter des Generator Network (Zeile 9-10).

```
1 noise2 = torch.randn(cur_batch_size, z_dim, device=device)
2 fake_2 = gen(noise2)
3
4 disc_fake_pred = disc(fake_2)
5
6 gen_loss = criterion(disc_fake_pred,
7                       torch.ones_like(disc_fake_pred))
8
9 gen_loss.backward()
10 gen_optimizer.step()
```

E: Training Loop von WGAN-GP

Das Konzept des Trainings von WGAN-GP ist dem von SGAN und DCGAN sehr ähnlich. Der einzige Unterschied ist, dass WGAN-GP mit W-Loss-Gp (*wasser stein loss with gradient Penalty*) trainiert wird. Außerdem benötigt der *Critic* nach den WGAN-GP Algorithmen *5 mal mehr* Training als *Generator* (siehe Alg.3). Critic und Generator werden 50 Mal mit dem gesamten Datensatz trainiert. Für jedes Mini-Batch wird Critic 5 Mal mehr als Generator trainiert. Bei jeder dieser 5 Iterationen wird ein Batch zufälliger Rauschvektoren abgetastet und an Generator gesendet, um gefälschte Bilder zu erzeugen. Dann werden die echten und gefälschten Bilder an Critic gesendet. Critic bewertet sie (vgl.Zeile 1-5 Alg.3). Vor der Berechnung von *Critic Loss* wird der *Gradient Penalty* berechnet (*Regularisierungs-Term*, vgl.4.3.3)

- **Berechnung von Gradient Penalty**
  1. Generierung eines Mini-Batch von Zufallsvektoren mit Werten zwischen `0` und `1` (*4D: Mini-Batch, Tiefe, Höhe, Breite*) für `epsilon` (vgl.Zeile 1 im Cod und Zeile 4 im Alg.3).
  2. Generierung eines Mini-Batch von *interpolierten Bildern* aus realen und gefälschten Bildern durch Multiplikation mit `epsilon` (vgl.Zeile 2 im Cod, Zeile 6 im Alg.3 und Abbildung 4.9).
  3. Die interpolierten Bilder werden von *Critic* bewertet und jedes Bild erhält einen Wert (vgl.Zeile 7 im Alg.3).
  4. Berechnung der Gradienten der Ausgaben (`critic_scores`) in Bezug auf die Eingaben (`interpolierte_images`) .

5. Berechnung der *L2-Norm* der Gradienten für jede Zeile (Größe von Gradienten)
6. Bestrafung des durchschnittlichen quadratischen Abstands der Gradienten-Normen von `1`.

```python
def gradient_penalty(critic, real, fake):
batch_size = len(real)
# a random vector between real and fake image dis in latent space
epsilon = torch.rand(batch_size, 1, 1, 1, device=device, requires_grad=
    True)

interpolated_images = real * epsilon + fake * (1 - epsilon)
critic_score = critic(interpolated_images)

gradient = torch.autograd.grad(
    outputs=critic_score,
    inputs=interpolated_images,
    grad_outputs=torch.ones_like(critic_score),
    create_graph=True,
    retain_graph=True,
)[0]

# flatten gradient = each image in one row
gradient = gradient.view(gradient.shape[0], -1)
# calculate the L2 norm of gradient
gradient_norm = gradient.norm(2, dim=1)
# panalizing the mean distance of the gradient from 1
gradient_penalty = torch.mean((gradient_norm - 1) ** 2)

return gradient_penalty
```

Listing 4.5: Gradient-Penalty

- **Berechnung von Critic Loss**

  Critic Loss ist die Differenz zwischen dem Mittelwert der *Critic Scores* (Bewertungen) für die gefälschten und die echten Bilder im Batch. Die Gradient Penalty wird mit *Lambda* multipliziert und zu dieser Differenz addiert (1-L-Continuity Enforcement, siehe u.A. 4.3.2). Da Critic in jeder Iteration 5 Mal trainiert wird, ist der endgültige Verlust der Mittelwert des Critic Loss in jeder Iteration geteilt durch 5.

```python
loss_critic = torch.mean(critic_fake) - torch.mean(critic_real) +
    LAMBDA_GP * gp
```

- **Berechnung von Generator Loss**

  Der Generator Loss ist das Negativ des Mittelwerts der Critic Scores (Bewertungen) für gefälschte Bilder.

```
gen_loss = (-1.) * torch.mean(critic_fake_score)
```

4.4.3 Speichern und Laden von Modell-Checkpoints

nach jeweils 500 Schritten, wenn die FID-Bewertung für das Modell niedriger ist als zuvor, werden die *Generator-* und *Discriminator-Modelle*, deren *Optimizers*, *Gewichten* und *Parametern* als *Dictionary* unter Verwendung von `state_dict()` (Python Dictionary Object) gespeichert. Dieses wird zum späteren Trainieren des Modells sowie für die *Inference* verwendet. Die Methode save_checkpoint() speichert jedes Checkpoint lokal in einem Verzeichnis.

```
if FID_value < old_fid:
    checkpoint = {
        'gen_state_dict': gen.state_dict(),
        'critic_state_dict': critic.state_dict(),
        'gen_optim_state_dict': gen_optimizer.state_dict(),
        'critic_optim_state_dict': critic_optimizer.state_dict(),
    }

# call save checkpoint method
save_checkpoint(checkpoint)

old_fid = FID_value
```

Durch das Laden von Checkpoints werden alle *Gewichte* und *Parameter* des Modells mit dem Modell geladen.

```
def load_checkpoint(checkpoint):
    gen.load_state_dict(checkpoint['gen_state_dict'])
    disc.load_state_dict(checkpoint['disc_state_dict'])
    gen_optimizer.load_state_dict(checkpoint['gen_optim_state_dict'])
    disc_optimizer.load_state_dict(checkpoint['disc_optim_state_dict'])

    # loading models on GPU (device='cuda')
    gen.to(device)
    disc.to(device)
```

4.4.4 Inference

Um das Modell für die **Inferenz** zu verwenden, wird es zunächst mit allen Gewichten und Parametern unter Verwendung von `state_dict()` geladen. Dann wird `gen.eval()` aufgerufen, um *Dropout- und Batch Normalization-Schichten* in den *Evaluierungsmodus* zu setzen. Schließlich wird die *Inference* ausgeführt, dann werden einige zufällige Rauschvektoren generiert und an das Modell übergeben. Das Modell wird nun gefälschte Gesichtsbilder generieren. Die generierten Bilder werden mit Hilfe von *Matplotlib* visualisiert und bei Bedarf gespeichert.

```
load_mode = True
if load_model:
    load_checkpoint(torch.load("my_checkpoint_DCG_test.pth.tar"))
    gen.eval()

for i in range(10):
    new_noise = torch.randn(batch_size, z_dim, 1, 1, device=device)
    new_fake_image = gen(new_noise)

    # plot and save images
    fake_img = new_fake_image
    show_tensor_images(fake_img, cur_step, state=True)
```

Listing 4.6: Inference

Zur *Visualisierung* der Bilder und der zugehörigen Daten in *TensorBoard* wird die *SummaryWriter-Bibliothek* verwendet (mehr dazu in 6.4). Die Implementierung im Zusammenhang mit der *Evaluierung* von GAN-Modellen wird in *Kapitel 5* erläutert.

4.4.5 Übersicht über die Modelle

Die folgenden Abbildungen veranschaulichen den Aufbau der Generator- und Discriminator-Modelle im Überblick.

generator model:

Layer (type)	Output Shape	Param #
ConvTranspose2d-1	[-1, 512, 4, 4]	819,200
BatchNorm2d-2	[-1, 512, 4, 4]	1,024
ReLU-3	[-1, 512, 4, 4]	0
ConvTranspose2d-4	[-1, 256, 8, 8]	2,097,152
BatchNorm2d-5	[-1, 256, 8, 8]	512
ReLU-6	[-1, 256, 8, 8]	0
ConvTranspose2d-7	[-1, 128, 16, 16]	524,288
BatchNorm2d-8	[-1, 128, 16, 16]	256
ReLU-9	[-1, 128, 16, 16]	0
ConvTranspose2d-10	[-1, 64, 32, 32]	131,072
BatchNorm2d-11	[-1, 64, 32, 32]	128
ReLU-12	[-1, 64, 32, 32]	0
ConvTranspose2d-13	[-1, 3, 64, 64]	3,072
Tanh-14	[-1, 3, 64, 64]	0

Total params: 3,576,704
Trainable params: 3,576,704
Non-trainable params: 0

Input size (MB): 0.00
Forward/backward pass size (MB): 3.00
Params size (MB): 13.64
Estimated Total Size (MB): 16.64

discriminator model:

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 32, 32]	3,136
BatchNorm2d-2	[-1, 64, 32, 32]	128
LeakyReLU-3	[-1, 64, 32, 32]	0
Conv2d-4	[-1, 128, 16, 16]	131,200
BatchNorm2d-5	[-1, 128, 16, 16]	256
LeakyReLU-6	[-1, 128, 16, 16]	0
Conv2d-7	[-1, 256, 8, 8]	524,544
BatchNorm2d-8	[-1, 256, 8, 8]	512
LeakyReLU-9	[-1, 256, 8, 8]	0
Conv2d-10	[-1, 512, 4, 4]	2,097,664
BatchNorm2d-11	[-1, 512, 4, 4]	1,024
LeakyReLU-12	[-1, 512, 4, 4]	0
Conv2d-13	[-1, 1, 1, 1]	8,193

Total params: 2,766,657
Trainable params: 2,766,657
Non-trainable params: 0

Input size (MB): 0.05
Forward/backward pass size (MB): 2.81
Params size (MB): 10.55
Estimated Total Size (MB): 13.41

Abbildung 4.10: DCGAN und WGAN-GP Modelle

generator model:

Layer (type)	Output Shape	Param #
Linear-1	[-1, 64]	8,256
BatchNorm1d-2	[-1, 64]	128
ReLU-3	[-1, 64]	0
Linear-4	[-1, 128]	8,320
BatchNorm1d-5	[-1, 128]	256
ReLU-6	[-1, 128]	0
Linear-7	[-1, 256]	33,024
BatchNorm1d-8	[-1, 256]	512
ReLU-9	[-1, 256]	0
Linear-10	[-1, 512]	131,584
BatchNorm1d-11	[-1, 512]	1,024
ReLU-12	[-1, 512]	0
Linear-13	[-1, 12288]	6,303,744
Sigmoid-14	[-1, 12288]	0

Total params: 6,486,848
Trainable params: 6,486,848
Non-trainable params: 0

Input size (MB): 0.00
Forward/backward pass size (MB): 0.21
Params size (MB): 24.75
Estimated Total Size (MB): 24.96

discriminator model:

Layer (type)	Output Shape	Param #
Linear-1	[-1, 256]	3,145,984
LeakyReLU-2	[-1, 256]	0
Linear-3	[-1, 128]	32,896
LeakyReLU-4	[-1, 128]	0
Linear-5	[-1, 64]	8,256
LeakyReLU-6	[-1, 64]	0
Linear-7	[-1, 1]	65

Total params: 3,187,201
Trainable params: 3,187,201
Non-trainable params: 0

Input size (MB): 0.05
Forward/backward pass size (MB): 0.01
Params size (MB): 12.16
Estimated Total Size (MB): 12.21

Abbildung 4.11: SGAN Modelle

5 Evaluation von GANs

Dieses Kapitel befasst sich mit der Evaluierung der von GAN Modellen erzeugten Bilder. In diesem Zusammenhang wird zunächst ein kurzer Überblick über die möglichen Methoden zur Evaluierung von GANs gegeben und anschließend werden die in dieser Arbeit verwendeten Evaluierungsmethoden ausführlicher erläutert.

5.1 GANs Problems

Die GAN-Modelle leiden beim Training unter vielen Problemen. Einige dieser Probleme sind unten beschrieben.

Mode Collapse

Dies tritt auf, wenn der Generator lernt, nur einige gute Samples zu erzeugen und den Discriminator mit diesen Samples zu täuschen. In diesem Fall konzentriert sich der Generator auf die Erzeugung einer begrenzten **Varietät** von Samples [36].

Abbildung 5.1: Mode Collapse während des Trainings von Standard-GAN Auf dem CelebA-Datensatz

Non-convergence
Ein weiterer negativer Aspekt ist, dass das Modell während des Trainings nicht stabil ist und sehr viel Zeit zum Trainieren benötigt. Manchmal schwanken die Modell-Parameter und das Modell konvergiert nie. Daher ist es erforderlich, häufig zu überprüfen, um festzustellen, wann das Training zu beenden ist [37].

Vanishing-Gradients
Es passiert, wenn der Discriminator zu erfolgreich wird, so dass der Generator-Gradient verschwindet und der Generator nichts mehr lernt (siehe Kapitel 4).

Evaluation
Die Messung der Leistung von GANs ist sehr herausfordernd und kompliziert. Der Grund dafür ist, dass bei der Bewertung von GANs verschiedene Faktoren und Aspekte wie Qualität, Diversität, Voreingenommenheit (Eng. biasing), Rechenleistung und ... berücksichtigt werden müssen. Deshalb ist es schwierig eine konkrete intrinsische Bewertungsmetrik zu finden, die all diese Aspekte abdeckt [4]. Dieses Thema wird im folgenden Abschnitt näher erläutert.

5.2 Maßstäbe zur Evaluierung von GANs

Die Evaluierung von GANs ist eine anspruchsvolle Herausforderung und es ist ein aktives Forschungsgebiet. Wie es bereits in Kapitel 3 erwähnt wurde, werden die Generator-Modelle nicht direkt trainiert, sondern mittels eines zweiten Modells, dem sogenannten Discriminator. Daher gibt es keine Zielfunktion oder kein objektives Maß für das Generator-Modell. Dies erschwert den Vergleich der Leistung dieser Modelle [4]. Andererseits erreicht der Discriminator, der den Generator trainiert, niemals die Perfektion, so dass er auch nicht für die Evaluierung verwendet werden kann. Denn ein sehr gut trainierter Discriminator bringt das Minimax-Spiel aus dem Gleichgewicht und gibt dem Generator keine Feedbacks zur Verbesserung (alle generierten Bilder als Fälschung klassifizieren), was den Generator überfordert (overfit).

Obwohl es viele Faktoren gibt, die bei der Bewertung von Gans berücksichtigt werden müssen aber die zwei Haupteigenschaften, nämlich **Fidelität** (Qualität der Bilder) und **Diversität** (Vielfalt der Bilder, z.B. alle Arten von Gesichtern, mit unterschiedlichen Haut-, Augen- und Haarfarben, Gefühlen, Alter und ...), sind besonders wichtig. Damit ist gemeint, dass ein gutes GAN-Modell nicht nur realistische Bilder mit hoher Qualität erzeugen können muss, sondern die erzeugten Bilder müssen auch eine hohe Vielfalt aufweisen. Daher konzentrieren sich fast alle bisher vorgestellten Maßstäbe zur Evaluierung von GANs hauptsächlich auf diese beiden Faktoren, um sie zu quantifizieren.

Nach Borji [4] lassen sich alle diese Maßstäbe in die folgenden zwei Kategorien einteilen:

- **Qualitativ**: Die qualitativen Maßstäbe sind nicht numerisch und beinhalten oft eine subjektive Bewertung durch den Menschen oder eine Bewertung durch Vergleich. In Abbildung 5.3 sind einige dieser Metriken mit deren Beschreibungen dargestellt.
- **Quantitativ**: Die quantitativen Maßstäbe sind weniger subjektiv und beziehen sich auf die Berechnung spezifischer numerischer Werte (oft ein einziger Wert). Sie entsprechen nicht direkt der Art und Weise, wie Menschen generierte Bilder wahrnehmen und beurteilen. Abbildung 5.2 stellt 24 verschiedene quantitative Maßstäbe für die Evaluierung von GANs dar.

	Measure	Description
Quantitative	1. Average Log-likelihood [18, 22]	• Log likelihood of explaining realworld held out/test data using a density estimated from the generated data (*e.g.* using KDE or Parzen window estimation). $L = \frac{1}{N}\sum_i \log P_{model}(\mathbf{x}_i)$
	2. Coverage Metric [33]	• The probability mass of the true data "covered" by the model distribution $C := P_{data}(dP_{model} > t)$ with t such that $P_{model}(dP_{model} > t) = 0.95$
	3. Inception Score (IS) [3]	• KLD between conditional and marginal label distributions over generated data. $\exp\left(\mathbb{E}_{\mathbf{x}}\left[\mathbb{KL}\left(p\left(\mathrm{y} \mid \mathrm{x}\right) \parallel p\left(\mathrm{y}\right)\right)\right]\right)$
	4. Modified Inception Score (m-IS) [34]	• Encourages diversity within images sampled from a particular category. $\exp(\mathbb{E}_{\mathbf{x}_i}[\mathbb{E}_{\mathbf{x}_j}[(\mathbb{KL}(P(y\|\mathbf{x}_i)\|P(y\|\mathbf{x}_j))]])$
	5. Mode Score (MS) [35]	• Similar to IS but also takes into account the prior distribution of the labels over real data. $\exp\left(\mathbb{E}_{\mathbf{x}}\left[\mathbb{KL}\left(p\left(y \mid \mathrm{x}\right) \parallel p\left(y^{train}\right)\right)\right] - \mathbb{KL}\left(p\left(y\right) \parallel p\left(y^{train}\right)\right)\right)$
	6. AM Score [36]	• Takes into account the KLD between distributions of training labels vs. predicted labels, as well as the entropy of predictions. $\mathbb{KL}(p(y^{\text{train}}) \parallel p(y)) + \mathbb{E}_{\mathbf{x}}\left[H(y\|\mathbf{x})\right]$
	7. Fréchet Inception Distance (FID) [37]	• Wasserstein-2 distance between multi-variate Gaussians fitted to data embedded into a feature space $FID(r,g) = \|\mu_r - \mu_g\|_2^2 + Tr(\Sigma_r + \Sigma_g - 2(\Sigma_r\Sigma_g)^{\frac{1}{2}})$
	8. Maximum Mean Discrepancy (MMD) [38]	• Measures the dissimilarity between two probability distributions P_r and P_g using samples drawn independently from each distribution. $M_k(P_r, P_g) = \mathbb{E}_{\mathbf{x},\mathbf{x}'\sim P_r}[k(\mathbf{x},\mathbf{x}')] - 2\mathbb{E}_{\mathbf{x}\sim P_r, y\sim P_g}[k(\mathbf{x},y)] + \mathbb{E}_{y,y'\sim P_g}[k(y,y')]$
	9. The Wasserstein Critic [39]	• The critic (*e.g.* an NN) is trained to produce high values at real samples and low values at generated samples $\hat{W}(\mathbf{x}_{test}, \mathbf{x}_g) = \frac{1}{N}\sum_{i=1}^{N}\hat{f}(\mathbf{x}_{test}[i]) - \frac{1}{N}\sum_{i=1}^{N}\hat{f}(\mathbf{x}_g[i])$
	10. Birthday Paradox Test [27]	• Measures the support size of a discrete (continuous) distribution by counting the duplicates (near duplicates)
	11. Classifier Two Sample Test (C2ST) [40]	• Answers whether two samples are drawn from the same distribution (*e.g.* by training a binary classifier)
	12. Classification Performance [1, 15]	• An indirect technique for evaluating the quality of unsupervised representations (*e.g.* feature extraction; FCN score). See also the GAN Quality Index (GQI) [41].
	13. Boundary Distortion [42]	• Measures diversity of generated samples and covariate shift using classification methods.
	14. Number of Statistically-Different Bins (NDB) [43]	• Given two sets of samples from the same distribution, the number of samples that fall into a given bin should be the same up to sampling noise
	15. Image Retrieval Performance [44]	• Measures the distributions of distances to the nearest neighbors of some query images (*i.e.* diversity)
	16. Generative Adversarial Metric (GAM) [31]	• Compares two GANs by having them engaged in a battle against each other by swapping discriminators or generators. $p(\mathrm{x}\|y=1;M_1^{'})/p(\mathrm{x}\|y=1;M_2^{'}) = \big(p(y=1\|\mathrm{x};D_1)p(\mathrm{x};G_2)\big)/\big(p(y=1\|\mathrm{x};D_2)p(\mathrm{x};G_1)\big)$
	17. Tournament Win Rate and Skill Rating [45]	• Implements a tournament in which a player is either a discriminator that attempts to distinguish between real and fake data or a generator that attempts to fool the discriminators into accepting fake data as real.
	18. Normalized Relative Discriminative Score (NRDS) [32]	• Compares n GANs based on the idea that if the generated samples are closer to real ones, more epochs would be needed to distinguish them from real samples.
	19. Adversarial Accuracy and Divergence [46]	• Adversarial Accuracy. Computes the classification accuracies achieved by the two classifiers, one trained on real data and another on generated data, on a labeled validation set to approximate $P_g(y\|\mathbf{x})$ and $P_r(y\|\mathbf{x})$. Adversarial Divergence: Computes $\mathbb{KL}(P_g(y\|\mathbf{x}), P_r(y\|\mathbf{x}))$
	20. Geometry Score [47]	• Compares geometrical properties of the underlying data manifold between real and generated data.
	21. Reconstruction Error [48]	• Measures the reconstruction error (*e.g.* L_2 norm) between a test image and its closest generated image by optimizing for z (*i.e.* $min_{\mathbf{z}}\|\|G(\mathbf{z}) - \mathbf{x}^{(test)}\|\|^2$)
	22. Image Quality Measures [49, 50, 51]	• Evaluates the quality of generated images using measures such as SSIM, PSNR, and sharpness difference
	23. Low-level Image Statistics [52, 53]	• Evaluates how similar low-level statistics of generated images are to those of natural scenes in terms of mean power spectrum, distribution of random filter responses, contrast distribution, etc.
	24. Precision, Recall and F_1 score [23]	• These measures are used to quantify the degree of overfitting in GANs, often over toy datasets.

Abbildung 5.2: Quantitative Maßstäbe zur Evaluierung von GANs nach Borji [4]

Qualitative	1. Nearest Neighbors	• To detect overfitting, generated samples are shown next to their nearest neighbors in the training set
	2. Rapid Scene Categorization [18]	• In these experiments, participants are asked to distinguish generated samples from real images in a short presentation time (*e.g.* 100 ms); *i.e.* real v.s fake
	3. Preference Judgment [54, 55, 56, 57]	• Participants are asked to rank models in terms of the fidelity of their generated images (*e.g.* pairs, triples)
	4. Mode Drop and Collapse [58, 59]	• Over datasets with known modes (*e.g.* a GMM or a labeled dataset), modes are computed as by measuring the distances of generated data to mode centers
	5. Network Internals [1, 60, 61, 62, 63, 64]	• Regards exploring and illustrating the internal representation and dynamics of models (*e.g.* space continuity) as well as visualizing learned features

Abbildung 5.3: Qualitative Maßstäbe zur Evaluierung von GANs nach Borji [4]

Alle diese Maßstäbe haben Vor- und Nachteile und es gibt keinen Konsens darüber, welcher Maßstab die Stärken und Schwächen von Modellen bestmöglich erfasst und für einen fairen Modell-Vergleich verwendet werden sollte. Die quantitativen Maßstäbe wurden von Borji In Abb 3 anhand folgender Kriterien verglichen:

- **Discriminability**: Die Fähigkeit, generierte Samples von echten Samples zu unterscheiden (hohe Fidelität) .
- **Detecting Overfitting**: Overfitting-Probleme treten auf, wenn der Discriminator zu stark von den Trainingsdaten abhängt. Wenn dieses Problem fortbesteht, ähnelt das vom Generator erzeugte Bild dem Lernbild (geringe Diversität) [38].
- **Disentangled Latent Spaces**: Es bietet die Möglichkeit, bestimmte Merkmale zu kontrollieren (z.B. Haarfarbe).
- **Well-defined Bounds**: Gut definierte Begrenzungen haben (untere, obere und zufällige)
- **Perceptual Judgments**: Wie nahe liegen ihre Bewertungen mit den menschlichen Beurteilungen und der menschlichen Bewertung für das Modell.
- **Sensitivity to Distortions**: Unveränderlichkeit bei Änderung der Größe von Daten (z.B. gleiches Ergebnis für ein Bild in verschiedenen Größen, oder Empfindlichkeit gegenüber Rauschen auf den generierten Bildern).
- **Comp. & Sample Efficiency**: Geringe Sample- und Berechnungskomplexität.

Measure		Desiderata: Discriminability	Detecting Overfitting	Disentangled Latent Spaces	Well-defined Bounds	Perceptual Judgments	Sensitivity to Distortions	Comp. & Sample Efficiency
1. Average Log- likelihood	[18, 22]	low	low	-	$[-\infty, \infty]$	low	low	low
2. Coverage Metric	[33]	low	low	-	[0, 1]	low	low	-
3. Inception Score (IS)	[3]	high	moderate	-	$[1, \infty]$	high	moderate	high
4. Modified Inception Score (m-IS)	[34]	high	moderate	-	$[1, \infty]$	high	moderate	high
5. Mode Score (MS)	[35]	high	moderate	-	$[0, \infty]$	high	moderate	high
6. AM Score	[36]	high	moderate	-	$[0, \infty]$	high	moderate	high
7. Fréchet Inception Distance (FID)	[37]	high	moderate	-	$[0, \infty]$	high	high	high
8. Maximum Mean Discrepancy (MMD)	[38]	high	low	-	$[0, \infty]$	-	-	-
9. The Wasserstein Critic	[39]	high	moderate	-	$[0, \infty]$	-	-	low
10. Birthday Paradox Test	[27]	low	high	-	$[1, \infty]$	low	low	-
11. Classifier Two Sample Test (C2ST)	[40]	high	low	-	[0, 1]	-	-	-
12. Classification Performance	[1, 15]	high	low	-	[0, 1]	low	-	-
13. Boundary Distortion	[42]	low	low	-	[0, 1]	-	-	-
14. NDB	[43]	low	high	-	$[0, \infty]$	-	low	-
15. Image Retrieval Performance	[44]	moderate	low	-	*	low	-	-
16. Generative Adversarial Metric (GAM)	[31]	high	low	-	*	-	-	moderate
17. Tournament Win Rate and Skill Rating	[45]	high	high	-	*	-	-	low
18. NRDS	[32]	high	low	-	[0, 1]	-	-	poor
19. Adversarial Accuracy & Divergence	[46]	high	low	-	[0, 1], $[0, \infty]$	-	-	-
20. Geometry Score	[47]	low	low	-	$[0, \infty]$	-	low	low
21. Reconstruction Error	[48]	low	low	-	$[0, \infty]$	-	moderate	moderate
22. Image Quality Measures	[49, 50, 51]	low	moderate	-	*	high	high	high
23. Low-level Image Statistics	[52, 53]	low	low	-	*	low	low	-
24. Precision, Recall and F_1 score	[23]	low	high	✓	[0, 1]	-	-	-

Abbildung 5.4: Vergleich von qualitativen Maßstäben zur Evaluierung von GANs nach Borji [4].'-' bedeutet unbekannt und '*' bezeichnet, dass mehrere verfügbar sind.

Auch wenn es schwierig ist, das perfekte Maßstab für die Bewertung aller Aspekte eines GAN-Modells mit nur einem einzigen Wert zu schaffen, aber zumindest können sie dazu verwendet werden, um zu evaluieren, welches GAN-Modell bessere Bilder erzeugt als andere.

Die in dieser Arbeit implementierten GAN-Modelle werden **quantitativ** und **qualitativ** evaluiert. Die ausgewählten Metriken zur quantitativen und quantitativen Evaluierung von GAN-Modellen werden in den folgenden Abschnitten näher erläutert.

5.3 Quantitative Evaluation der mit DCGAN, WGAN-GP generierten Bilder

Nach Borji erscheint **Fréchet Inception Distance (FID)** plausibler als andere Maßstäbe. Außerdem stimmt FID mit menschlichen Urteilen überein und ist robuster gegenüber Rauschen im Vergleich zu anderen Metriken wie **Inception Score (IS)** (vgl.Abbildung 5.4). Anhand dieser Gründe wurde FID für die quantitative Evaluierung dieser Modelle ausgewählt (siehe Abbildung 5.5).

Mathematisch gesehen ist **Fréchet Distance** auch bekannt als *Wasserstein-2 Distance* eine Metrik, mit der der Abstand zwischen zwei *multivariate Normalverteilungen* berechnet wird [39]. Diese Metrik wird in FID verwendet, um den Abstand zwischen der generierten Verteilung und der realen Verteilung zu messen [3]. FID betrachtet den *Mittelwert* und die *Kovarianzmatrix* der realen und gefälschten multivariate Normalverteilungen und berechnet, wie weit diese Statistiken voneinander entfernt sind. Die FID-Gleichung für eine multivariate Normalverteilung ist wie folgt.

$$d^2(X, Y) = \|\mu_X - \mu_Y\|^2 + Tr\left[\Sigma_X + \Sigma_Y - 2(\Sigma_X\Sigma_Y)^{1/2}\right] \tag{5.1}$$

wobei:

X, Y *Reale und gefälschte Normalverteilungen*

μ_X *Mittelwert*

Tr Trace der Matrix (die Summe der Diagonalelemente einer Matrix)

Σ *Kovarianzmatrix*

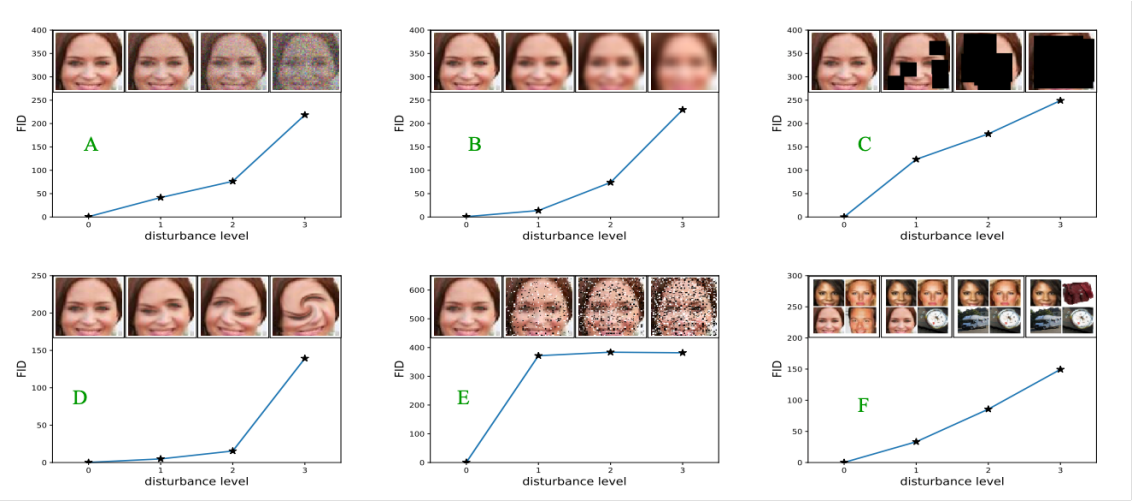


Abbildung 5.5: FID ist empfindlich gegenüber A: Gaußsches Rauschen, B: Gaußsche Unschärfe, C: implantierte schwarze Rechtecke, D: verwirbelte Bilder, E: Salz- und Pfefferrauschen und F: CelebA-Datensatz kontaminiert durch ImageNet-Bilder [3].

Für die quantitative Evaluierung von realen und generierten Gesichtsbildern verwendet FID den vortrainierten **Inception-v3 Classifier** [3]. Dieser Classifier wurde auf einem riesigen Datensatz namens ImageNet trainiert [40]. Dieser Datensatz enthält etwa 14 Millionen Bilder in 20 tausend Kategorien. Dies gibt dem Inception-v3 Classifier die Möglichkeit, zu viele gute Features aus dieser riesigen Vielfalt an Bildern zu extrahieren. Daher enthält die letzte Pooling-Schicht des Inceptionv3 Classifier zu viele nützliche Features, die für das Feature-Matching und die Erkennung von High Level Features (z.B.: verschiedene Arten von Augen, Lippen, Nasen und ... Paterns) in einem Bild verwendet werden können [41].

Die quantitative Evaluierung von GAN-Modellen durch die FID-Methode wird wie folgt durchgeführt:

- Das vortrainierte Inception-v3-Modell wird ohne die letzte Schicht (*fully connected layer* übernimmt die Klassifizierungsaufgabe) aus dem `torch Modul` geladen. Das inception-v3 Modell hat 42 Schichten mit einer Eingabegröße von `299x299x3` und die Ausgabe der letzten pooling Schicht ist `8x8x2048` (vgl.Abbildung 5.6). Die Eingabebilder müssen auf einen Bereich von `[0, 1]` mit dem `Mittelwert = [0,485, 0,456, 0,406]` und dem `Standardwert = [0,229, 0,224, 0,225]` normalisiert werden.
- Ein *Mini-Batch* von realen und generierten Bildern wird in die Eingabegröße des Inception-v3-Modells transformiert und an das Modell zur Feature-Extraktion weitergeleitet, was auch als *Embedding* bezeichnet wird (Übersetzung von hochdimensionalen Vektoren in den niedrig-dimensionalen Raum, d.h. ein großer Bildrepräsentationsvektor wird in der letzten Pooling-Schicht in einen Vektor mit `2048` Werten übersetzt (*Embedding*)).

- Die Ausgabe der letzten Pooling-Schicht für jedes reale und jedes generierte Bild im Inception-v3-Modell ist ein Vektor mit `2048` Zahlen oder Dimensionen. Dieser Vektor enthält alle Merkmale dieses Bildes und wird als Embedding bezeichnet. Wenn also ein Mini-Batch von `100` realen oder gefälschten Bildern an das Inception-v3-Modell übergeben wird, besteht die Ausgabe aus `100x2048` Vektoren, die eine multivariate Normalverteilung bilden (einer für reale und einer für generierte Bilder). Die FID berechnet den Abstand zwischen diesen beiden Verteilungen. Je niedriger die **FID-Score**, desto geringer ist der Abstand und desto ähnlicher sind sich die generierten und die realen Bilder, während eine höhere Score bedeutet, dass sie sehr unterschiedlich sind.

type	patch size/stride or remarks	input size
conv	3×3/2	299×299×3
conv	3×3/1	149×149×32
conv padded	3×3/1	147×147×32
pool	3×3/2	147×147×64
conv	3×3/1	73×73×64
conv	3×3/2	71×71×80
conv	3×3/1	35×35×192
3×Inception	As in figure 5	35×35×288
5×Inception	As in figure 6	17×17×768
2×Inception	As in figure 7	8×8×1280
pool	8 × 8	8 × 8 × 2048
linear	logits	1 × 1 × 2048
softmax	classifier	1 × 1 × 1000

Abbildung 5.6: Die Grundzüge der Inception-v3 Netzwerk-Architektur [40]

5.3.1 Implementierung von FID

Die Umsetzung stellt sich wie folgt dar:

1. Laden des vortrainierten Inception-v3-Modells

Zunächst wird das Inception-Modell aus dem Torchvision-Modul geladen. Da das vortrainierte Modell benötigt wird, wird der Parameter `pretrained` auf `True` gesetzt. Nach dem Laden wird es so eingestellt, dass es auf der GPU (`device='cuda'`) läuft, und der Evaluierungsmodus wird aktiviert (`inception_model.eval()`).

```
1 import torchvision.models as models
2 inception_model = models.inception_v3(pretrained=True)
3 inception_model.to(device)
```

```
4 inception_model= inception_model.eval()
```

2. Entfernen der letzten Schicht (fc) des Inception-Modells

Das *Fully Connected Layer (fc)* des `inception_model` wird durch die *Identity Layer(Identit())* ersetzt. Das Identity Layer ist wie ein Platzhalter und gibt seine Eingaben unverändert zurück.

```
1 inception_model.fc = torch.nn.Identity()
```

3. Berechnung der Frechet-Inception-Distanz (FID)

Um eine Verlangsamung der Trainingsleistung zu vermeiden, wird die FID alle *500 Iterationen* berechnet. Zur Berechnung von FID sind folgende Methoden implementiert.

A: Größenanpassung von Bildern

Bevor die realen und generierten Bilder zur Feature-Extraktion an `inception_model` übergeben werden, müssen sie die gleiche Größe wie die `inception_model` Eingabe (`299x299x3`) haben. Dies erfolgt durch die Implementierung der Funktion `incep_size_img(image)` . Diese Methode erhält ein *Mini-Batch* von realen oder gefälschten Bildern als Eingabe und liefert veränderte Bilder in der Größe von `299x299x3` .

```
def incep_size_img(image):
    img = torch.nn.functional.interpolate(image, size=(299, 299),
                                          mode='bilinear',
                                          align_corners=False)
    return img
```

B: Feature-Extraktion

Nach der Größenanpassung der realen und gefälschten Bilder werden diese zur Merkmalsextraktion an `inception_model` gesendet.

Dies geschieht in der Funktion `extract_features(real_images, fake_images)` . Die extrahierten Features werden in zwei verschiedenen Listen (`real_extracted_features` , `fake_extracted_features`) gespeichert. Diese Listen werden im nächsten Schritt für die Berechnung des *Mittelwerts* ($\mu = mu$) und der *Kovarianzen* ($\Sigma = sigma$) verwendet. Sie werden später ebenfalls paarweise mit Hilfe von *Pairplot* visualisiert.

```
def extract_features(real_images, fake_images) :

    real_samples = incep_size_img (real_images)
    fake_samples = incep_size_img (fake_images)

    features_real = inception_model(real_samples.to(device)).detach().to('cpu')
    real_extracted_features.append(features_real)

    features_fake = inception_model(fake_samples.to(device)).detach().to('cpu')
    fake_extracted_features.append(features_fake)
```

C: Berechnung von Mittelwert und Kovarianz

Die extrahierten Features für reale und gefälschte Bilder bilden zwei *multivariate Normalverteilungen*. In der folgenden Methode werden alle gespeicherten Features in der Liste `real_extracted_features` , und der Liste `fake_extracted_features` zusammengefügt

und zwei große *Tensors* gebildet. Diese beiden Tensors werden für die Berechnung des *Mittelwerts* ($\mu = mu$) und der *Kovarianz* ($\Sigma = sigma$) der echten und der gefälschten Verteilung verwendet. Die von dieser Methode zurückgegebenen *Mittelwerte* (μ_x, μ_y) und *Kovarianzen* (Σ_x, Σ_y) werden im nächsten Schritt für die Berechnung der **Fréchet Distance** verwendet.

```
1  def cal_mu_and_sigma():
2
3      all_real_features = torch.cat(real_extracted_features)
4      all_fake_features = torch.cat(fake_extracted_features)
5
6      mu_real = all_real_features.mean(0)
7      mu_fake = all_fake_features.mean(0)
8
9      sigma_real = cal_covariance(all_real_features)
10     sigma_fake = cal_covariance(all_fake_features)
11
12     return mu_real, mu_fake, sigma_real, sigma_fake
```

Die Berechnung der Kovarianzen erfolgt mit der Methode `cal_covariance(features)`.

```
1  def cal_covariance(features):
2      return torch.Tensor(np.cov(features.detach().numpy(), rowvar=False))
```

D: Berechnung der Multivariate Fréchet Distance

Die Methode `rechet_distance(mu_x, mu_y, sigma_x, sigma_y)` nimmt die *Mittelwerte* und *Kovarianzen* aus **C:** und berechnet die **Fréchet Distance** zwischen den realen und gefälschten Verteilungen (vgl. Gleichung 5.1).

```
1  def frechet_distance(mu_x, mu_y, sigma_x, sigma_y):
2
3      first_term = (mu_x - mu_y).dot(mu_x - mu_y)
4
5      sigma_num = (sigma_x @ sigma_y).cpu().detach().numpy()
6      sigma_sqr = scipy.linalg.sqrtm(sigma_num)
7      sigma_tens = torch.Tensor(sigma_sqr.real)
8
9      second_term = torch.trace(sigma_x) + torch.trace(sigma_y) - 2*torch.trace(
       sigma_tens)
10
11     f_distance = first_term + second_term
12
13     return f_distance
```

5.4 Qualitative Evaluation der mit DCGAN, WGAN-GP generierten Bilder

Die qualitative Evaluierung von GAN-Modellen basiert auf dem Prinzip der **Rapid Scene Categorization** (siehe Abbildung 5.3). Denn eine der üblichen und intuitiven Methoden zur Bewertung von GANs ist die **visuelle Untersuchung** [4].
Bei dieser Methode werden den Probanden die generierten Bilder für eine bestimmte Zeit vorgeführt, um zu überprüfen, ob sie die generierten Bilder von echten Bildern unterscheiden können. Ein Beispiel für eine solche qualitative Bewertung ist in der Arbeit von Denton u.a. (**LAPGAN**) [42] zu sehen.

In der oben genannten Arbeit wurden deren GAN-Modelle (LAPGAN, CC-LAPGAN) und ein Standard-GAN-Modell (Goodfellow et al. 2014) auf dem **CIFAR10-Datensatz** trainiert. Für die qualitative Bewertung dieser 3 Modelle wurden die generierten Bilder und auch die realen CIFAR10-Bilder (4 Arten von Bildern) den 15 Freiwilligen unter Verwendung einer Benutzeroberfläche gezeigt. Die realen CIFAR10-Bilder wurden den Testpersonen vor dem Experiment gezeigt. Die Präsentationszeiten lagen zwischen 50 ms und 2000ms. Die Probanden wurden gebeten, die generierten Bilder von den echten Bildern zu unterscheiden, indem sie auf der Benutzeroberfläche auf die passende Taste (real, fake) klicken.

Die qualitative Bewertung der implementierten GAN-Modelle in dem vorliegenden Arbeit stützt sich ebenfalls auf eine ähnliche Methode. Das heißt, nach dem Training von SGAN, DCGAN und WGAN auf dem **CelebA-Datensatz** wird zuerst eine bestimmte Anzahl von generierten Gesichtsbildern ausgewählt. Dann werden die Bilder über eine Benutzeroberfläche (**Image Viewer**) den Versuchspersonen präsentiert. Die Testpersonen werden gebeten, die generierten Gesichtsbilder von echten Gesichtsbildern in einer bestimmten Zeit zu unterscheiden und ihre Urteile über die integrierten Tasten (real, fake) der Benutzeroberfläche (Image viewer) abzugeben.

5.4.1 Implementierung von Image Viewer

Zur Präsentation der generierten Gesichtsbilder für die Probanden ist ein Image viewer implementiert. Für die Implementierung wurde ein Python-GUI-Framework namens **Tkinter** verwendet. Der Image viewer hat ein einfaches Design. Das Bild wird im oberen Bereich angezeigt und zwei Tasten befinden sich unterhalb des Bildes (siehe Abbildung 5.7).

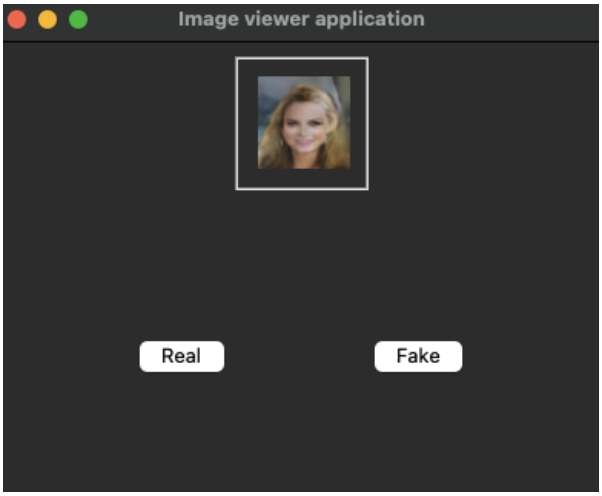


Abbildung 5.7: Image viewer

Nach dem Ausführen von `image_viewer.py` werden die Namen und Pfade aller Bilder in einer 2D-Liste (`image_list`) gespeichert. Außerdem werden vor der Darstellung der Bilder für die Testpersonen die Elemente der 2D-Liste gemischt. Dann wird der Image viewer geöffnet und das erste Bild (Start-Bild) wird dem Probanden angezeigt (siehe Abbildung 5.7). Zur gleichen Zeit beginnt der timer, die Zeit zu berechnen. Beim klicken auf den real oder fake Button werden zwei Ereignisse ausgelöst (`combine_funcs(count_time('real'), next_image())` oder `combine_funcs(count_time('fake'), next_image())`).

Die Methode `combine_funcs(*funcs)` ruft die Methode `count_time(which_button)` auf, um die verbrachte Zeit zu berechnen und das Urteil (real oder fake) zu speichern. Sie ruft außerdem die Methode `next_image()` auf, um das nächste Bild zu zeigen und zu überprüfen, ob das Bild nicht länger als die Experimentzeit betrachtet wurde. Wenn die Betrachtungszeit nicht größer als die Experimentzeit ist, wird der Name des Bildes mit dem Urteil in einer anderen Liste(`img_after_study`) gespeichert. Andererseits, wenn die verbrachte Zeit größer als die erlaubte Zeit ist, bleibt das Bild in der Liste und die Versuchsperson sieht es am Ende wieder und beurteilt es erneut. Nachdem alle Bilder vom Probanden beurteilt wurden, kann die neue Liste, die alle Namen und Beurteilungen der Bilder enthält, als *CSV-Datei* gespeichert werden, um sie später für die Auswertung zu verwenden.

Das Skript `calculaty_votes.py` liest die *CSV-Datei* und berechnet die Anzahl der echten stimmen für jedes Modell. Das Ergebnis wird anschließend durch *matplotlib* veranschaulicht und lokal gespeichert.

6 Tools und Experimente

In diesem Kapitel werden zunächst alle in dieser Arbeit verwendeten Tools und Frameworks vorgestellt. Anschließend werden zwei durchgeführte Experimente und deren Ergebnisse beschrieben.

6.1 Dataset

Im Rahmen dieser Arbeit wurde der CelebFaces-attributes-Dataset (CelebA)[1] benutzt. Dieser ist ein großer Datensatz von Gesichtsattributen mit mehr als 200.000 Bildern von Prominenten. Die Bilder in diesem Datensatz decken große Posenvariationen und Hintergrundunschärfen ab [44].
Der Datensatz umfasst 10.177 Identitäten, 202.599 Gesichtsbilder und 5 Orientierungspunkte mit jeweils 40 binären Attributen pro Bild.
Die große Vielfalt, die große Quantität und die zahlreichen Annotationen machen diesen Datensatz zu einer guten Wahl für verschiedene Computer-Vision-Aufgaben, einschließlich Gesichtsbearbeitung und -synthese (Eng. face editing & synthesis).

6.2 Entwicklungsumgebung

Diese Arbeit basiert auf einer **Remote-Entwicklungsumgebung**. Um diese Umgebung bereitzustellen, wurden folgende Schritte durchgeführt:

1. **Verbindungsaufbau mit dem Remote-Server**: um eine Verbindung zwischen dem Mac-Gerät und dem Remote-Server der technischen Hochschule Brandenburg von außen herzustellen, werden „**Fortclient**“ (VPN-Software) und das integrierte „**Terminal**“ in **Visual-Studio-Code** (SSH-Verbindung) verwendet.

2. **Aufbau und Aktivierung einer isolierten virtuellen Umgebung auf dem Remote Server**: auf einem server sind normalerweise viele verschiedene Programme, Frameworks und Bibliotheken mit unterschiedlichen Versionen und Abhängigkeiten (Eng. dependencies) installiert. außerdem können dort viele Prozesse gleichzeitig laufen. Diese können zu einem Konflikt führen. Zur Verhinderung eines solchen konfliktes wird eine isolierte virtuelle Umgebung mit einer Python-Bibliothek namens „**virtualenv**“ erstellt und aktiviert. In dieser virtuellen Umgebung wurde **Python 3.7.5** benutzt.

[1] Der CelebA-Datensatz, der in dieser Arbeit verwendet wird, wurde von Kaggle.com heruntergeladen [43].

3. **Installation der erforderlichen APIs**: nach der Erstellung einer isolierten virtuellen Umgebung auf dem Server wurden alle benötigten Frameworks, Bibliotheken und APIs für diese Arbeit wie **Jupyter-Notbook**, **Pytorch**, **TensorBaord** und ... in dieser Umgebung installiert.

4. **Ausführen von Jupyter-Notebook über einen Web-Browser**: die Jupyter-Notebook-App ist eine Server-Client-Anwendung, die das Bearbeiten und Ausführen von Notebook-Dokumenten über einen Webbrowser ermöglicht. Die Notebook-Dokumente sind von Menschen lesbare Dokumente, die von der Jupyter Notebook App erzeugt werden. Sie enthalten Computercode (z.B. Python) und Rich-Text-Elemente (Absätze, Gleichungen, Abbildungen, Links, etc...). Das Starten von Jupyter-Notbooks über einen Web-Browser wird mit einem einzigen Befehl im Terminal ermöglicht.

   ```
   jupyter notebook --no-browser --ip=xxx --port xxx
   ```

Für den gesamten Implementierungsteil dieser Abschlussarbeit wird das **Jupyter-Notbook** und die Programmiersprache **Python** benutzt. Die Auswahl von Python für diese Arbeit basiert auf folgenden Gründen:

- Die Entwicklung in Python ist prägnant, lesbar und bietet eine einfache Handhabung und Simplizität [45].
- Python bietet eine umfangreiche Auswahl an Bibliotheken und Frameworks für Machine-Learning-Projekte wie NumPy, Pandas, Matplotlib, SciPy, TensorFlow, Pytorch und mehr.
- Python hat eine große, aktive Community und es gibt eine Fülle von Unterstützung.

Das Training der implementierten GAN-Modelle wird auf einer **NVIDIA Titan RTX** Grafikkarte mit **CUDA 11.1** durchgeführt.
In den folgenden Abschnitten werden der grundlegende Aufbau und die Benutzeroberfläche von Jupyter-Notebook etwas näher betrachtet.

6.2.1 Jupyter-Notebook

Das Jupyter-Notebook ist eine Web-Applikation für die Erstellung und den Austausch von Berechnungsdokumenten [46].
Der Aufbau des Jupyter-Notebook besteht aus zwei Komponenten:
Eine Web-Applikation: ein Browser-basiertes Werkzeug für die interaktive Erstellung von Dokumenten, die Erklärungstexte, Mathematik, Berechnungen und deren Rich-Media-Ausgabe kombinieren.
Notebook-Dokumente: eine Darstellung aller in der Web-Applikation sichtbaren Inhalte, einschließlich Eingaben und Ausgaben von Berechnungen, Erklärungstext, Mathematik, Bildern und Rich-Media-Darstellungen von Objekten.
Die mit Jupyter-Notebook erstellten Dokumente sind in der Regel JSON-Dateien und werden mit der Erweiterung `.ipynb` gespeichert.

6.2.2 Notbook Benutzeroberfläche

Die Notebook-Umgebung ist einfach, übersichtlich und dokumentenzentrisch. Es besteht aus Notebook-Name, Menü-Bar, Tool-Bar und Zellen (siehe Abbildung 6.1).

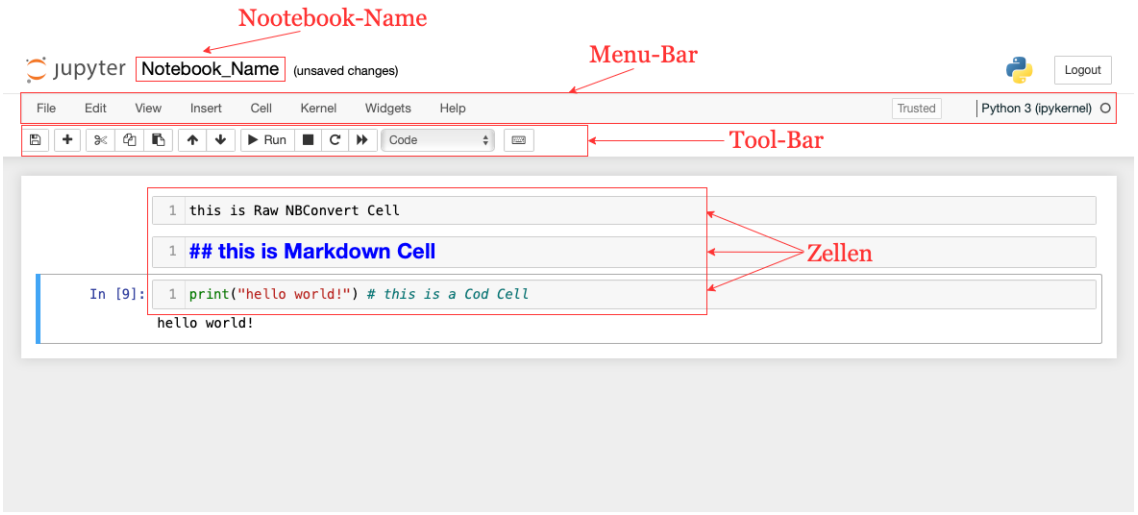


Abbildung 6.1: Übersicht einer Notebook-Umgebung

Das Notebook hat drei Arten von Zellen:

- **Raw-NBConvert-Zelle**: es kann zum Umwandeln von verschiedenen Codeformaten im HTML- oder LaTeX-Format durch Sphinx verwendet werden.
- **Markdown-Zelle**: es nutzt die Markup-Sprache „Markdown“ und rendert den Inhalt als HTML und dient als Erklärungstext.
- **Code-Zelle**: diese Art von Zelle wird zum Schreiben von Cod-Syntaxen (z. B. Python-Code) verwendet. Der Inhalt wird zur Ausführung an den Kernel[2] gesendet und die Ausgabe wird in der Fußzeile der Zelle angezeigt.

Das Notebook besteht aus einer Folge von Zellen. Die Inhalte der Zellen lassen sich durch Drücken der Tasten **Shift + Enter** oder durch Drücken von „**Run**“ in der Tool-Bar oder von „**Cell**“ in der Menü-Bar ausführen.

6.3 Frameworks und Bibliotheken

In der Arbeit werden folgende Python-Bibliotheken verwendet:

[2]Ein Notebook-Kernel ist eine "Berechnungsmaschine", die den in einem Notebook-Dokument enthaltenen Code ausführt (z. B. ipykernel für Python) [46].

- **Pytorch**
  Pytorch ist ein weit verbreitetes Framework für maschinelles Lernen, das vom Facebook AI Research Team entwickelt wurde. Diese Open-Source-optimierte Tensor-Bibliothek für Deep Learning kann sowohl auf GPUs als auch auf CPUs laufen. Dieses ML-Framework wird für die Umsetzung des GANs-Modells in der vorliegenden Arbeit benutzt.

- **Seaborn und Matplotlib**
  Seaborn und Matplotlib werden zur Datenvisualisierung benutzt.

- **NumPy**
  Numpy wird für einige Operationen mit N-dimensionalen Arrays auf der CPU eingesetzt.

- **Pandas**
  Pandas ist eine Open-Source-Bibliothek, die leistungsfähige Strukturen und Werkzeuge wie DataFrames und Series für die Datenanalyse in der Programmiersprache Python bereitstellt. Diese Bibliothek wird zur Erstellung einer Dataframe-Struktur aus echten und gefälschten Bildern und zur Verkettung dieses Dataframes benutzt.

- **Torchvision**
  Diese Bibliothek ist ein Bestandteil des PyTorch-Projekts. Das torchvision-Package besteht aus populären Datensätzen, Modellarchitekturen und gängigen Bildtransformationen für Computer Vision. Diese wird zum Laden und Transformieren des Datensatzes sowie zur Visualisierung der Daten verwendet.

6.4 Visualisierungsumgebung

Für die Visualisierung von Workflow, Metriken, Graphen und Bildern der in dieser Arbeit implementierten Modelle wird das **TensorBoard** verwendet.
TensorBoard ist eine Visualisierungserweiterung, die vom TensorFlow-Team entwickelt wurde, um die Komplexität von neuronalen Netzen zu reduzieren [47].
Dieses Visualisierungs-Toolkit von TensorFlow bietet viele Werkzeuge und Funktionen, einige davon sind wie folgt:

- Verfolgung und Visualisierung von Metriken wie Verlust (loss) und Genauigkeit(accuracy)
- Visualisierung des Modellgraphen (Ops und Schichten)
- Darstellung der zeitlichen Entwicklung von Gewichten, Biases oder anderen Tensoren in Form von Histogrammen
- Projizieren von Embeddings[3] in einen niedriger-dimensionalen Raum

[3] Ein Embeddings ist ein verhältnismäßige niedrig-dimensionaler Raum, in den die hochdimensionalen Vektoren übersetzt werden können. [48].

- Visualisierung von Bildern, Text und Audiodaten

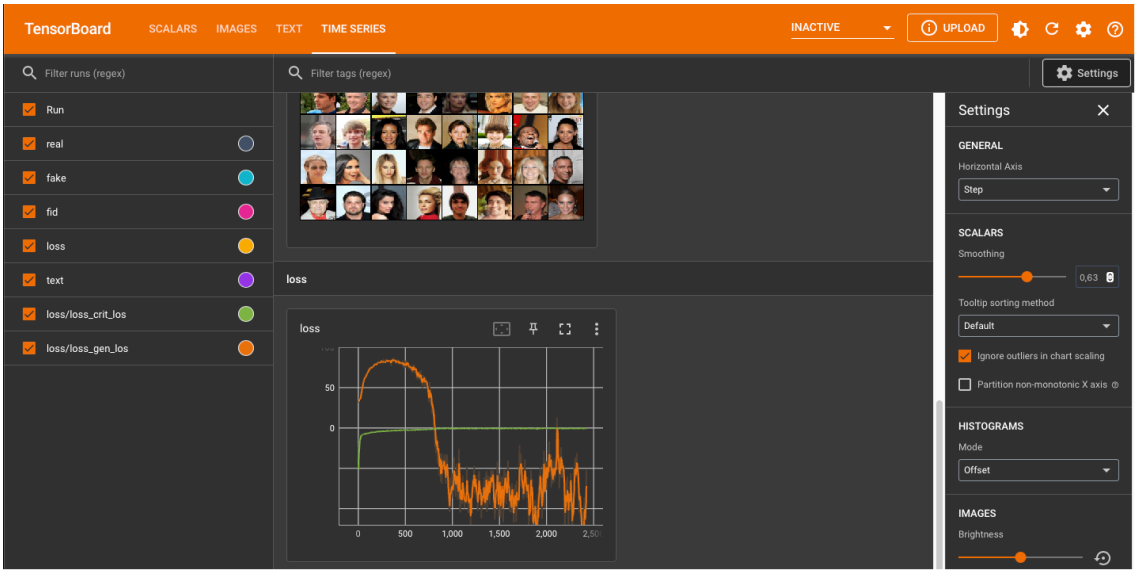


Abbildung 6.2: Übersicht von TensorBoard-Dashboard

6.4.1 Einsatz von TensorBoard im Zusammenhang mit Pytorch

wie es bereits in Abschnitt 3 erwähnt wurde, wird Pytorch als Machine Learning Framework für diese Arbeit benutzt. Andererseits ist TensorBoard eigentlich ein TensorFlow-Visualisierungs-Toolkit. Um diese Erweiterung in Pytorch zu verwenden, werden folgende Schritte durchgeführt.

Installation von TensorBoard in der erstellten Umgebung
Für die Installation dieses Visualisierungs-Toolkit in der aktivierten, erstellten Umgebung, wird das Python-Paketverwaltungssystem **pip** verwendet.
Installation von TensorBoard über die Kommandozeile:

```
pip install tensorboard
```

Im Jupyter-Notebook:

```
!pip install tensorboard
```

Installation von Pytorch für die Verwendung des SummaryWriter Moduls
PyTorch bietet die Möglichkeit, Modelle und Metriken in das TensorBoard Log-Verzeichnis zu protokollieren (loggen). Diese Möglichkeit wird durch die Klasse **SummaryWriter** in Pytorch

angeboten. Diese Klasse stellt eine High-Level API zur Erstellung einer Log-Datei(event file) in einem bestimmten Verzeichnis zur Verfügung und fügt dort Summaries und Events hinzu. Die Klasse aktualisiert den Inhalt der Datei asynchron, wodurch der Trainingsprozess nicht verlangsamt wird [49].
In der Kommandozeile:

```
pip install torch
```

Im Jupyter-Notebook:

```
!pip install torch
```

Bevor irgendetwas in das TensorBoard Log-Verzeichnis protokolliert wird, wird es nötig sein, eine SummaryWriter Instanz zu erstellen. Diese Instanz (`writer`) speichert standardmäßig die Logdatei im Verzeichnis `./runs/`.

```
import torch
from torch.utils.tensorboard import SummaryWriter
writer = SummaryWriter()
```

Beispiel:

```
import torch
from torch.utils.tensorboard import SummaryWriter
import numpy as np

# the write instance saves the log files in ./test_log_directory
writer = SummaryWriter(f"./test_log_directory")

for n_iter in range(20):
    writer.add_scalar('Loss', np.random.random(), n_iter)
```

Ausführen von TensorBoard
Um das TensorBoard zu starten, wird der folgende Befehl in der Kommandozeile eingegeben. Dieser Befehl liefert eine URL, die das TensorBoard in einem Browser öffnet.

```
# in Command Line
tensorboard --logdir=runs

# in Jupyter Notbook
!tensorboard --logdir=runs
```

6.5 Experimente und Ergebnisse

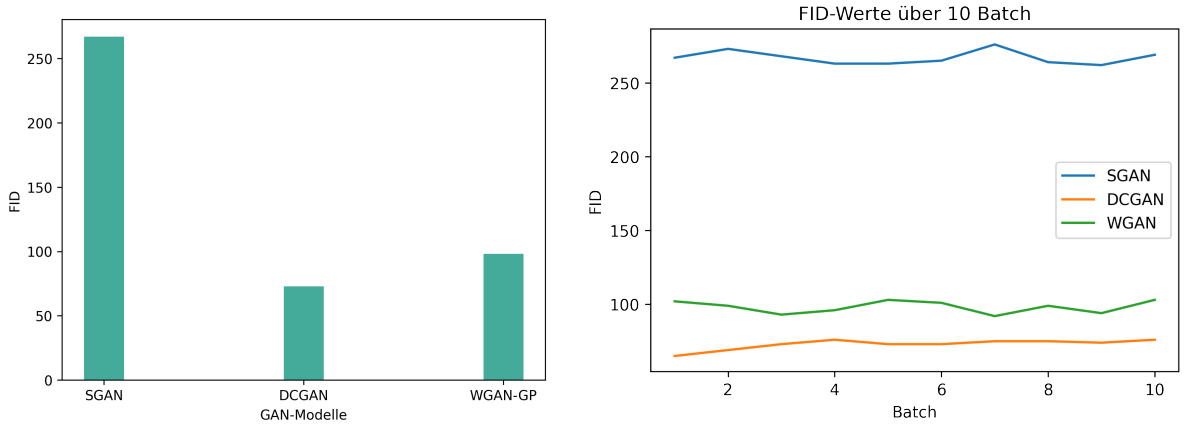
Die Qualität der generierten Bilder ist sehr wichtig und spielt daher die Hauptrolle bei der Evaluierung der GAN-Modelle. Im folgenden Abschnitt werden zwei Experimente durchgeführt, um die Qualität der generierten Bilder nach dem Training der GAN-Modelle zu messen.

6.5.1 Experiment 1 – Quantitative Evaluierung der durch SGAN, DCGAN und WGAN-GP generierten Bilder unter Verwendung von FID

Das Ziel dieses Experiments besteht darin, die Qualität der von SGAN, DCGAN und WGAN-GP erzeugten Bilder nach dem Training dieser Modelle quantitativ zu vergleichen. Dies erfolgt unter Verwendung der FID-Metrik (vgl. Abschn. 5.3).

Zur Durchführung dieses Experiments werden zunächst die trainierten Modelle SGAN, DCGAN und WGAN-GP geladen. Nach dem Laden der Modelle werden 1280 Bilder von jedem Modell generiert. Diese 1280 Bilder werden in 10 Iterationen generiert. Die Qualität der generierten Bilder in jeder Iteration wird durch die FID-Metrik gemessen. Da die GAN-Modelle stochastische Eigenschaften haben, wird für den endgültigen FID-Wert für jedes GAN-Modell das arithmetische Durchschnitt der FID-Werte aus diesen 10 Iterationen berücksichtigt.

Das Ergebnis dieses Experiments lässt sich wie folgt darstellen.



(a) DCGAN hat niedrigerer FID-Wert.

(b) Geringe Änderungen der FID-Werte nach dem Training (Modellstabilität)

Abbildung 6.3: Das Ergebnis der quantitativen Evaluierung von GAN-Modellen, Ein niedrigerer FID-Wert entspricht einer besseren Qualität der Bilder.

Nach den Ergebnissen des Experiments hat DCGAN niedrigere FID-Werte und erzeugt Bilder mit besserer Qualität als SGAN und WGAN-GP (vgl.Abbildung 6.3). Außerdem hat SGAN sehr hohe FID-Werte, was bedeutet, dass die Qualität der von diesem Modell erzeugten Bilder sehr niedrig ist. Die FID-Werte von WGAN-GP liegen nahe an den FID-Werten von DCGAN. Dies bedeutet, dass der *WGP-Loss* die Qualität der erzeugten Bilder nicht wesentlich verbessert.

6.5.2 Experiment 2 – Qualitative Evaluierung der durch SGAN, DCGAN und WGAN-GP generierten Bilder

Das Ziel dieses Experiments ist es, die von SGAN, DCGAN und WGAN-GP erzeugten Bilder nach dem Training qualitativ (auf menschlicher Basis) zu bewerten. Dies wird entsprechend der in Abschnitt 5.4 beschriebenen Methode durchgeführt.

Für die Durchführung dieses Experiments werden zunächst von jedem Modell 30 Bilder unter Verwendung eines Zufallsrauschens erzeugt (insgesamt 90 Bilder). Jedes dieser Bilder wird unter einem bestimmten Namen gespeichert, aus dem hervorgeht, von welchem Modell es erzeugt wurde. Außerdem werden 30 Bilder aus realen Daten (Datensatz) zufällig ausgewählt und ebenfalls unter bestimmten Namen gespeichert.

Zur Präsentation dieser Bilder für Testpersonen wird der implementierte *Image Viewer* (siehe u.A. 5.4.1) verwendet. Der Image Viewer mischt die echten und gefälschten Bilder (120 echte und gefälschte Bilder und ein Startbild) vor der Darstellung. Anschließend werden die 10 Probanden aufgefordert, innerhalb von 5 Sekunden zwischen echten und gefälschten Bildern zu unterscheiden. Das Ergebnis dieses Experiments ist in Abbildung 6.4 dargestellt.

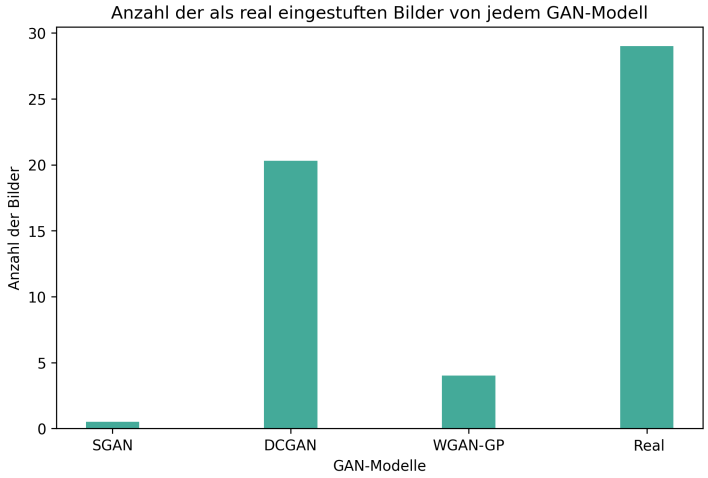
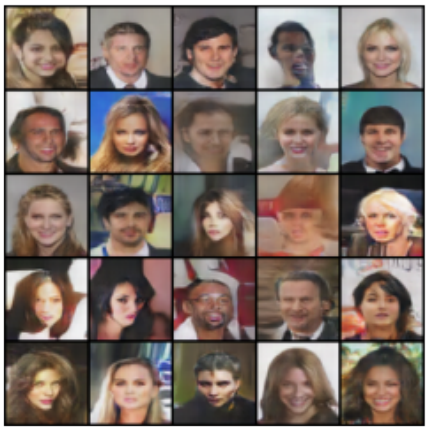


Abbildung 6.4: Das Ergebnis der qualitativen Evaluierung von GAN-Modellen

Das Ergebnis der qualitativen Bewertung von SGAN, DCGAN und WGAN-GP bestätigt ebenfalls, dass DCGAN Bilder mit besserer Qualität als die anderen beiden Modelle erzeugt

(vgl. Abbildung 6.4 und Abbildung 6.3). Der wichtige Punkt dieses Experiments ist, dass das Ergebnis von DCGAN sehr nah an den realen Daten liegt. Das bedeutet, dass es den Testpersonen schwerer fiel, die von DCGAN generierten Bilder von den realen Bildern zu unterscheiden. SGAN leidet unter Mode Collapse und konvergiert nicht (vgl. Abbildung 6.5). Ein weiterer Punkt ist, dass das Ergebnis von DCGAN fast fünfmal besser als das von WGAN-GP ist. Wenn man dies mit dem Ergebnis der quantitativen Auswertung vergleicht, wird deutlich, dass der Unterschied dort nicht so groß ist. Dies kann damit erklärt werden, dass bei der quantitativen Auswertung auch menschliche Verzerrungen (Eng. human biasing) eine wichtige Rolle spielen. Abbildung 6.4 zeigt auch, dass nicht alle echten Bilder von den Testpersonen als echt bewertet wurden. Die generierten Bilder in Abbildung 6.5 verdeutlichen die Ergebnisse der obigen Experimente.



(a) DCGAN.

(b) WGAN-GP.

(c) SGAN.

(d) Real.

Abbildung 6.5: Die generierten Bilder durch a: DCGAN, b: WGAN-GP, c: SGAN Modelle neben d: Reale Bilder aus dem CelebA-Datensatz

Abbildungsverzeichnis

Tabellenverzeichnis

List of Algorithms

Listings

Literatur

[1] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville und Y. Bengio. *Generative Adversarial Networks*. 2014. arXiv: 1406.2661 [stat.ML].

[2] T. Karras, M. Aittala, S. Laine, E. Härkönen, J. Hellsten, J. Lehtinen und T. Aila. „Alias-Free Generative Adversarial Networks“. In: *Proc. NeurIPS*. 2021.

[3] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler und S. Hochreiter. „GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium“. In: *Advances in Neural Information Processing Systems*. Hrsg. von I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan und R. Garnett. Bd. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper/2017/file/8a1d694707eb0fefe65871369074926d-Paper.pdf.

[4] A. Borji. *Pros and Cons of GAN Evaluation Measures*. 2018. arXiv: 1802.03446 [cs.CV].

[5] C. Aggarwal. *Neural Networks and Deep Learning: A Textbook*. Springer International Publishing, 2018, S. 1. ISBN: 9783319944630. URL: https://books.google.de/books?id=achqDwAAQBAJ.

[6] C. E. IBM. *What are neural networks?* Aug. 2020. URL: https://www.ibm.com/cloud/learn/neural-networks (besucht am 03. 11. 2021).

[7] J. Brownlee. *Deep Learning With Python: Develop Deep Learning Models on Theano and TensorFlow Using Keras*. Machine Learning Mastery, 2016, S. 23–27. URL: https://books.google.de/books?id=K-ipDwAAQBAJ.

[8] S. Sharma. *What the hell is Perceptron?* Sep. 2017. URL: https://towardsdatascience.com/what-the-hell-is-perceptron-626217814f53 (besucht am 04. 11. 2021).

[9] J. Collis. *Glossary of deep learning: Bias*. Apr. 2017. URL: https://medium.com/deeper-learning/glossary-of-deep-learning-bias-cf49d9c895e2 (besucht am 07. 11. 2021).

[10] „Appendix A - Artificial neural networks“. In: *Neural Networks Modeling and Control*. Hrsg. von E. N. Sanchez, J. D. Rios, A. Y. Alanis, N. Arana-Daniel und C. Lopez-Franco. Academic Press, 2020, S. 117–124. ISBN: 978-0-12-817078-6. DOI: https://doi.org/10.1016/B978-0-12-817078-6.00016-7. URL: https://www.sciencedirect.com/science/article/pii/B9780128170786000167.

[11] A. Souza und F. Soares. *Neural Network Programming with Java*. Community experience distilled. Packt Publishing, 2016, S. 45–57. ISBN: 9781785880902. URL: https://de.scribd.com/book/342442737/Neural-Network-Programming-with-Java.

[12] P. Pragati Baheti. *12 types of neural networks activation functions: How to choose?* URL: https://www.v7labs.com/blog/neural-networks-activation-functions (besucht am 07. 11. 2021).

[13] I. Goodfellow, Y. Bengio und A. Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016, S. 80, 177, 326–328.

[14] J. TORRES. *Learning process of a deep neural network*. März 2021. URL: https://towardsdatascience.com/learning-process-of-a-deep-neural-network-5a9768d7a651 (besucht am 09. 11. 2021).

[15] C. Mc. *Machine learning fundamentals (i): Cost functions and gradient descent*. Nov. 2017. URL: https://towardsdatascience.com/machine-learning-fundamentals-via-linear-regression-41a5d11f5220 (besucht am 09. 11. 2021).

[16] G. Zaccone, M. Karim und A. Menshawy. *Deep Learning with TensorFlow*. Packt Publishing, 2017, S. 65–71. ISBN: 9781786460127. URL: https://de.scribd.com/book/382268771/Deep-Learning-with-TensorFlow.

[17] A. M. Tonello, N. A. Letizia, D. Righini und F. Marcuzzi. „Machine Learning Tips and Tricks for Power Line Communications“. In: *IEEE Access* 7 (2019), S. 82434–82452. DOI: 10.1109/ACCESS.2019.2923321.

[18] J. Brownlee. *Generative Adversarial Networks with Python: Deep Learning Generative Models for Image Synthesis and Image Translation*. Machine Learning Mastery, 2019, S. 4–7. URL: https://books.google.de/books?id=YBimDwAAQBAJ.

[19] D. Foster. *Generative Deep Learning: Teaching Machines to Paint, Write, Compose, and Play*. O'Reilly Media, 2019, S. 1–4. ISBN: 9781492041894. URL: https://books.google.de/books?id=RqegDwAAQBAJ.

[20] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville und Y. Bengio. „Generative Adversarial Nets“. In: *Advances in Neural Information Processing Systems*. Hrsg. von Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence und K. Q. Weinberger. Bd. 27. Curran Associates, Inc., 2014. URL: https://proceedings.neurips.cc/paper/2014/file/5ca3e9b122f61f8f06494c97b1afccf3-Paper.pdf.

[21] A. Katz und E. Ross. *Minimax*. URL: https://brilliant.org/wiki/minimax/ (besucht am 23. 12. 2021).

[22] X. Mao und Q. Li. *Generative Adversarial Networks for Image Generation*. Springer Singapore, 2021, S. 2–4. ISBN: 9789813360488. URL: https://books.google.de/books?id=u9oeEAAAQBAJ.

[23] I. Goodfellow. *NIPS 2016 Tutorial: Generative Adversarial Networks*. 2017. arXiv: 1701.00160 [cs.LG].

[24] D. Foster. *Generative Deep Learning: Teaching Machines to Paint, Write, Compose, and Play*. O'Reilly Media, 2019. ISBN: 9781492041894. URL: https://books.google.de/books?id=RqegDwAAQBAJ.

[25] S. Santurkar, D. Tsipras, A. Ilyas und A. Madry. *How Does Batch Normalization Help Optimization?* 2019. arXiv: 1805.11604 [stat.ML].

[26] S. Ioffe und C. Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG].

[27] S. Ioffe. *Batch Renormalization: Towards Reducing Minibatch Dependence in Batch-Normalized Models*. 2017. arXiv: 1702.03275 [cs.LG].

[28] A. Radford, L. Metz und S. Chintala. *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*. 2016. arXiv: 1511.06434 [cs.LG].

[29] P. R. Matthew Stewart. *Simple introduction to Convolutional Neural Networks*. Juni 2020. URL: https://towardsdatascience.com/simple-introduction-to-convolutional-neural-networks-cdf8d3077bac (besucht am 02. 01. 2022).

[30] V. Dumoulin und F. Visin. *A guide to convolution arithmetic for deep learning*. 2018. arXiv: 1603.07285 [stat.ML].

[31] V. Dumoulin und F. Visin. „A guide to convolution arithmetic for deep learning“. In: *ArXiv e-prints* (März 2016). eprint: 1603.07285.

[32] J. Hui. *Gan - ways to improve gan performance*. März 2020. URL: https://towardsdatascience.com/gan-ways-to-improve-gan-performance-acf37f9f59b (besucht am 04. 01. 2022).

[33] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin und A. C. Courville. „Improved Training of Wasserstein GANs“. In: *Advances in Neural Information Processing Systems*. Hrsg. von I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan und R. Garnett. Bd. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper/2017/file/892c3b1c6dccd52936e27cbd0ff683d6-Paper.pdf.

[34] A. Dhinakaran. *Using statistical distance metrics for machine learning observability*. Okt. 2020. URL: https://towardsdatascience.com/using-statistical-distance-metrics-for-machine-learning-observability-4c874cded78 (besucht am 05. 01. 2022).

[35] M. Arjovsky, S. Chintala und L. Bottou. *Wasserstein GAN*. 2017. arXiv: 1701.07875 [stat.ML].

[36] K. Ahirwar. *Generative Adversarial Networks Projects: Build next-generation generative models using TensorFlow and Keras*. Packt Publishing, 2019, S. 32–33. ISBN: 9781789134193. URL: https://books.google.de/books?id=kmSGDwAAQBAJ.

[37] J. Hui. *Gan - why it is so hard to train generative adversarial networks!* Juni 2018. URL: https://jonathan-hui.medium.com/gan-why-it-is-so-hard-to-train-generative-advisory-networks-819a86b3750b (besucht am 05. 01. 2022).

[38] J. Kim und H. Park. „Limited discriminator GAN using explainable AI model for overfitting problem“. In: *ICT Express* (2022). ISSN: 2405-9595. DOI: https://doi.org/10.1016/j.icte.2021.12.014. URL: https://www.sciencedirect.com/science/article/pii/S240595952100179X.

[39] D. Dowson und B. Landau. „The fréchet distance between multivariate normal distributions“. In: *Journal of Multivariate Analysis* 12.3 (1982), S. 450–455. DOI: 10.1016/0047-259x(82)90077-x.

[40] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens und Z. Wojna. „Rethinking the Inception Architecture for Computer Vision“. In: *CoRR* abs/1512.00567 (2015). arXiv: 1512.00567. URL: http://arxiv.org/abs/1512.00567.

[41] *Generative Adversarial Networks (Gans) specialization*. URL: https://www.deeplearning.ai/program/generative-adversarial-networks-gans-specialization/ (besucht am 05. 01. 2022).

[42] E. L. Denton, S. Chintala, a. szlam arthur und R. Fergus. „Deep Generative Image Models using a Laplacian Pyramid of Adversarial Networks“. In: *Advances in Neural Information Processing Systems*. Hrsg. von C. Cortes, N. Lawrence, D. Lee, M. Sugiyama und R. Garnett. Bd. 28. Curran Associates, Inc., 2015. URL: https://proceedings.neurips.cc/paper/2015/file/aa169b49b583a2b5af89203c2b78c67c-Paper.pdf.

[43] J. Li. *Celebfaces attributes (celeba) dataset*. Jan. 2018. URL: https://www.kaggle.com/jessicali9530/celeba-dataset (besucht am 08. 11. 2021).

[44] Z. Liu, P. Luo, X. Wang und X. Tang. „Deep Learning Face Attributes in the Wild“. In: *Proceedings of International Conference on Computer Vision (ICCV)*. Dez. 2015.

[45] J. Protasiewicz. *Python ai: Why is python so good for machine learning?* Dez. 2021. URL: https://www.netguru.com/blog/python-machine-learning (besucht am 27. 12. 2021).

[46] *jupyter-notebook*. URL: https://jupyter-notebook.readthedocs.io/en/stable/notebook.html (besucht am 27. 12. 2021).

[47] *Get started with tensorboard*. URL: https://www.tensorflow.org/tensorboard/get_started?hl=en (besucht am 28. 12. 2021).

[48] *Embeddings*. URL: https://developers.google.com/machine-learning/crash-course/embeddings/video-lecture (besucht am 28. 12. 2021).

[49] *Torch.utils.tensorboard*. URL: https://pytorch.org/docs/stable/tensorboard.html (besucht am 29. 12. 2021).