

# **Einsatz von synthetischen Bilddaten zum Trainieren von Künstlichen Neuronalen Netzen zur Klassifizierung am Beispiel von Zellbildern**

## **Studienarbeit**

im Rahmen der Prüfung zum  
**Bachelor of Science (B. Sc.)**

des Studiengangs Informatik  
an der Dualen Hochschule Baden-Württemberg Karlsruhe

von

**Niklas Kemper**

Matrikelnummer, Kurs: 1799733, TINF19B2

und

**Fabian Schwickert**

Matrikelnummer, Kurs: 4439027, TINF19B4

Abgabedatum: 15.05.2022

Bearbeitungszeitraum: 04.10.2021 - 15.05.2022

Gutachter der Dualen Hochschule: Markus Reischl

# Eidesstattliche Erklärung

Ich versichere hiermit, dass ich meine Studienarbeit mit dem Thema:

*Einsatz von synthetischen Bilddaten zum Trainieren von Künstlichen Neuronalen Netzen zur Klassifizierung am Beispiel von Zellbildern*

gemäß § 5 der „Studien- und Prüfungsordnung DHBW Technik“ vom 01. Dezember 2019 selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Karlsruhe, den 15. Mai 2022

---

Niklas Kemper

Karlsruhe, den 15. Mai 2022

---

Fabian Schwickert

## **Abstract**

Der Einsatz von synthetischen Bilddaten ist eine Möglichkeit, neue Datensätze zu erstellen oder kleine Datensätze zu vergrößern, sodass Künstliche Neuronale Netze mit größeren und hochwertigeren Datensätzen trainiert werden können. In dieser Studienarbeit wurde untersucht, ob die Qualität von synthetischen Bilddaten, welche mit einem GAN generiert wurden, vergleichbar ist mit der Qualität von echten Bilddaten. Dafür wurden im ersten Schritt verschiedene GAN-Ansätze entwickelt, um möglichst hochwertige Bilddaten zu generieren. Im Anschluss wurden unterschiedliche Klassifizierungs-Netze mit diesen Bilddaten trainiert. Die Leistung der Netze wurde dann verglichen, wobei Unterschieden zwischen Training mit echten Daten, Training mit synthetischen Daten und Training mit gemischten Daten. Es stellte sich heraus, dass die Qualität der synthetischen Bilddaten nicht die Qualität der echten Bilddaten erreicht und die Leistung des Klassifizierungs-Netzes dementsprechend schlechter ist, als beim Training mit echten Bilddaten. Das gemischte Verwenden von synthetischen und echten Bilddaten stellte sich allerdings als funktionierende Methode heraus, um den Datensatz zu vergrößern.

# Inhaltsverzeichnis

<b>Abkürzungsverzeichnis</b>	<b>IV</b>
<b>Abbildungsverzeichnis</b>	<b>V</b>
<b>Tabellenverzeichnis</b>	<b>VII</b>
<b>1. Einleitung</b>	<b>1</b>
<b>2. Stand der Technik</b>	<b>4</b>
2.1. Künstliche Neuronale Netze . . . . .	4
2.2. Klassifizierung von Bilddaten (CNNs) . . . . .	20
2.3. Generative Adversarial Networks . . . . .	29
2.4. Synthetische Trainingsdaten mit GANs generieren . . . . .	42
<b>3. Konzepte der Studienarbeit</b>	<b>45</b>
3.1. Design-Aspekte von KNNs . . . . .	45
3.2. Konzept für die GANs . . . . .	46
3.3. Konzept für die Klassifizierungs-KNNs . . . . .	58
<b>4. Umsetzung und Implementierung</b>	<b>70</b>
4.1. Technische Hilfsmittel . . . . .	70
4.2. Vorbereiten der Laufzeitumgebung . . . . .	73
4.3. Implementierung der GANs . . . . .	75
4.4. Implementierung der Klassifiziererungs-KNNs . . . . .	78
<b>5. Ergebnisse</b>	<b>86</b>
5.1. Vorgehen und Bewertungs-Metriken . . . . .	86
5.2. Synthetischen Daten . . . . .	88
5.3. Testergebnisse der Klassifizierungs-KNNs . . . . .	92
<b>6. Diskussion und Bewertung</b>	<b>99</b>
<b>7. Fazit und Ausblick</b>	<b>104</b>
<b>Literaturverzeichnis</b>	<b>VIII</b>
<b>A. Anhang</b>	<b>XIV</b>
A.1. Hyperparameter-Suche für Klassifizierungs-KNNs . . . . .	XIV
A.2. Testergebnisse der Klassifizierungs-KNNs . . . . .	XX

# Abkürzungsverzeichnis

<b>ADAM</b>	Adaptive Moment Estimation
<b>API</b>	Application Programming Interface
<b>BCE</b>	Binary Cross Entropy
<b>CGAN</b>	Conditional Generative Adversarial Network
<b>CNN</b>	Convolutional Neural Network
<b>Colab</b>	Colaboratory
<b>FID</b>	Fréchet Inception Distance
<b>GAN</b>	Generative Adversarial Network
<b>ILSVCR</b>	ImageNet Large Scale Visual Recognition Challenge
<b>IncNet</b>	Inception Neural Network
<b>JSON</b>	JavaScript Object Notation
<b>KI</b>	Künstliche Intelligenz
<b>KNN</b>	Künstliches Neuronales Netz
<b>MSE</b>	Mean Squared Error
<b>ReLU</b>	Rectified Linear Unit
<b>ResNet</b>	Residual Neural Network
<b>RMSprop</b>	Root Mean Square Propagation
<b>VRAM</b>	Video Random Access Memory
<b>WCGAN</b>	Wasserstein-CGAN
<b>WGAN</b>	Wasserstein-GAN

# Abbildungsverzeichnis

2.1. Die Schichten-Struktur eines mehrschichtigen Künstliches Neuronales Netz (KNN) . . . . .	5
2.2. Der Aufbau eines künstlichen Neurons . . . . .	8
2.3. Ein Überblick über gängige Aktivierungsfunktionen . . . . .	11
2.4. Das Gradientenabstiegs-Verfahren mit integrierten Trägheitseffekt . . . . .	19
2.5. Bilddaten für die Verarbeitung mit KNNs formatieren . . . . .	23
2.6. Der Filterungsprozess einer Faltungsschicht . . . . .	25
2.7. Der Verdichtungsprozess einer Bündelungsschicht . . . . .	28
2.8. Die Funktionsweise eines Generative Adversarial Network (GAN) . . . . .	30
2.9. Minimax-Spiele mit Gradientenabstiegs-Verfahren bearbeiten . . . . .	34
2.10. Divergenz der Parameter bei Gradientenabstiegs-Verfahren im Kontext von Minimax-Spielen . . . . .	36
2.11. Die Architektur eines Conditional Generative Adversarial Network (CGAN) .	40
2.12. Funktionsweise der transponierten Faltung . . . . .	41
3.1. Architektur des Generators . . . . .	48
3.2. Parameter des Generator-Netzwerks . . . . .	51
3.3. Architektur des Diskriminators . . . . .	52
3.4. Der Mode Collapse des Generators nach zwei Trainings-Epochen . . . . .	54
3.5. Parameter des Diskriminator-Netzwerks . . . . .	55
3.6. Ein Inception Modul mit Bottleneck-Schichten . . . . .	60
3.7. Das Residual Neural Network (ResNet)-Konzept . . . . .	61
3.8. Die eigenen Schichten für die Klassifizierung . . . . .	62
5.1. Mit dem BCE-CGAN generierte Bilder von gesunden Zellen . . . . .	88
5.2. Mit dem BCE-CGAN generierte Bilder von kranken Zellen . . . . .	89
5.3. Mit dem LS-CGAN generierte Bilder von gesunden Zellen . . . . .	89
5.4. Mit dem LS-CGAN generierte Bilder von kranken Zellen . . . . .	90
5.5. Mit dem Wasserstein-CGAN (WCGAN) generierte Bilder von gesunden Zellen	90
5.6. Mit dem WCGAN generierte Bilder von kranken Zellen . . . . .	91

A.1. Vergleich der Loss-Funktionen für Klassifizierungs-KNNs . . . . .	XV
A.2. Vergleich der Trainings-Verfahren für Klassifizierungs-KNNs . . . . .	XVI
A.3. Vergleich der Lernraten für Klassifizierungs-KNNs . . . . .	XVII
A.4. Vergleich der Batch-Größen für Klassifizierungs-KNNs . . . . .	XVIII
A.5. Vergleich der Gewichts-Initialisierung für Klassifizierungs-KNNs . . . . .	XIX
A.6. Testergebnisse eines auf echten Daten trainierten Klassifizierungs-KNN . . . . .	XXI
A.7. Testergebnisse eines auf 10.000 mit einem Wasserstein-CGAN generierten Da- ten trainierten Klassifizierungs-KNN . . . . .	XXII
A.8. Testergebnisse eines auf 20.000 mit einem Wasserstein-CGAN generierten syn- thetischen Daten trainierten Klassifizierungs-KNN . . . . .	XXIII
A.9. Testergebnisse eines auf 30.000 mit einem Wasserstein-CGAN generierten syn- thetischen Daten trainierten Klassifizierungs-KNN . . . . .	XXIV
A.10. Testergebnisse eines auf 10.000 mit einem Binary Cross Entropy (BCE)-CGAN generierten synthetischen Daten trainierten Klassifizierungs-KNN . . . . .	XXV
A.11. Testergebnisse eines auf 20.000 mit einem BCE-CGAN generierten syntheti- schen Daten trainierten Klassifizierungs-KNN . . . . .	XXVI
A.12. Testergebnisse eines auf 30.000 mit einem BCE-CGAN generierten syntheti- schen Daten trainierten Klassifizierungs-KNN . . . . .	XXVII
A.13. Testergebnisse eines auf 10.000 mit einem Wasserstein-CGAN generierten syn- thetischen und 10.000 echten Daten trainierten Klassifizierungs-KNN . . . . .	XXVIII
A.14. Testergebnisse eines auf 20.000 mit einem Wasserstein-CGAN generierten syn- thetischen und 10.000 echten Daten trainierten Klassifizierungs-KNN . . . . .	XXIX
A.15. Testergebnisse eines auf 10.000 mit einem BCE-CGAN generierten syntheti- schen und 10.000 echten Daten trainierten Klassifizierungs-KNN . . . . .	XXX
A.16. Testergebnisse eines auf 20.000 mit einem BCE-CGAN generierten syntheti- schen und 10.000 echten Daten trainierten Klassifizierungs-KNN . . . . .	XXXI

# **Tabellenverzeichnis**

4.1. Vergleich PyTorch und Tensorflow . . . . .	73
5.1. FID-Scores der unterschiedlichen GANs . . . . .	92

# 1. Einleitung

Künstliche Intelligenz (KI) und maschinelles Lernen haben in den letzten Jahren viel Aufmerksamkeit auf sich gezogen und das Gebiet wächst stetig weiter. Ein Teilgebiet des maschinellen Lernens ist das „Deep Learning“. Beim Deep Learning werden KNNs eingesetzt, um Muster in Daten zu erkennen. Die Struktur und Funktionsweise eines KNN ähnelt der des menschlichen Gehirns. Wie bei einem menschlichen Gehirn besteht das KNN aus vielen Neuronen, die untereinander verknüpft sind. Durch diese Verknüpfungen können komplexe Sachverhalte modelliert werden. KNNs sind in der Lage, auf der Basis von Daten viele Probleme zu lösen, die mit herkömmlichen Algorithmen nur schwer oder gar nicht lösbar sind. Verbreitete Anwendungsgebiete von KNNs sind Predictive Maintenance, Bilderkennung und Prozessoptimierung [1, 2].

Ein Aufgabengebiet der KI ist die „Klassifizierung“. Bei der Klassifizierung wird versucht, ein in eingegebenen Daten abgebildetes Objekt einer bestimmten Klasse zuzuordnen. Dabei können die Daten in verschiedenen Formaten vorliegen. In den meisten Fällen handelt es sich um Bilddateien oder direkt ein Tupel mit den Merkmalen des zu klassifizierenden Objekts. Vor allem die Klassifizierung von Bildern ist ein Problem, das für Menschen leicht lösbar ist, Computer aber vor eine große Herausforderung stellt. Menschen sind sogar in der Lage, den Inhalt eines Bilds korrekt zu erkennen, wenn Teile des Bilds fehlen oder das Bild gedreht ist [2, 3].

Um ein KNN aufzubauen, werden große Datenmengen benötigt. Mit diesen Daten wird das Netz trainiert, sodass anschließend auch unbekannte Eingaben korrekt klassifiziert werden können. Durch eine Große Menge an Daten soll eine möglichst hohe Varianz innerhalb der Daten erreicht werden. Am Beispiel von Bilddaten meint diese Varianz z. B. unterschiedliche Blickwinkel auf Objekte, eine teilweise Überdeckung, das Fehlen von Teilen des Bilds und vieles mehr. Durch diese Varianz in den Daten sollen KNNs genau wie der Mensch auch unter unterschiedlichen Umwelteinflüssen stehende Objekte erkennen können [2, 4, 5].

Für viele Aufgabenstellungen sind jedoch keine passenden Datensätze zum Trainieren eines KNNs vorhanden. Qualitativ hochwertige Datensätze können oft nur mit hohen

Kosten erworben werden. Passende Datensätze selbst zu erheben, kann sich je nach Problemstellung ebenfalls als aufwändig und teuer erweisen [4, 5]. Deshalb bietet es sich an, synthetische Datensätze zu erzeugen.

Eine neue Lösung, um synthetische Trainingsdaten zu erzeugen, sind Generative Adversarial Networks (GANs). Die Idee eines GANs wurde zuerst im Jahr 2014 von Ian Goodfellow formuliert [6]. Ein GAN besteht aus einem Generator und einem Diskriminatoren. Der Generator erzeugt auf Grundlage zufälliger Startwerte ein synthetisches Daten-Objekt. Der Diskriminatoren ist ein binärer Klassifizierer, der entscheidet, ob ein Daten-Objekt echt oder synthetisch ist. Je ähnlicher die synthetischen Daten den echten Daten sind, desto ungenauer wird die Entscheidung des Diskriminators und desto hochwertiger sind die generierten Daten [7].

Durch den Einsatz eines GANs sinkt der Aufwand für die Beschaffung von passenden Datensätzen und die Varianz der Daten ist theoretisch höher [8]. GANs haben enormes Potential, weshalb Yann LeCun – ein führender Forscher auf dem Gebiet des maschinellen Lernens – GANs auch als „die coolste Idee des maschinellen Lernens in den letzten 20 Jahren“ bezeichnet [6].

Das Ziel der vorliegenden Studienarbeit ist es zu untersuchen, inwiefern sich mithilfe eines GANs generierte Bilder dazu eignen, ein KNN für die Klassifizierung von Bildern zu trainieren. Dafür werden KNNs miteinander verglichen, die ausschließlich auf echten, ausschließlich auf synthetischen oder einer Mischung aus beidem trainiert werden. Es soll untersucht und bewertet werden, wie GANs aufgebaut sind und welche Parameter gewählt werden müssen, um geeignete synthetische Bilddaten zu erzeugen. Die auf synthetischen Datensätzen trainierten Klassifizierungs-KNNs sollen eine hohe Vorhersage-Genauigkeit beim Test mit echten Bildern aufweisen und im besten Fall sogar genauer sein, als die mit den echten Datensätzen trainierten KNNs.

Diese Untersuchung wird am Beispiel von menschlichen Zellbildern durchgeführt. Zellbilder wurden ausgewählt, da es im menschlichen Körper über 200 unterschiedliche Zelltypen gibt [9] und nur wenige qualitativ hochwertige Datensätze von Zellbildern frei verfügbar sind. Damit steht die Forschung in diesem Bereich vor dem Problem, dass Trainings-Daten fehlen. Der Einsatz von KNNs in diesem Bereich ist besonders interessant, da KNNs die Analyse von Zellen unterstützen könnten. Vor allem die Entscheidung, ob eine Zelle gesund oder krank ist, lässt sich oft schwierig treffen. KNNs können

bei dieser Entscheidungsfindung eine zentrale Rolle einnehmen und so bei der Erkennung von Krankheiten helfen [10].

Im nachfolgenden Kapitel 2 werden zunächst die Grundlagen vermittelt, die für das Verständnis der vorliegenden Studienarbeit benötigt werden. Das umfasst eine Einführung in die Funktionsweise von KNNs (Unterkapitel 2.1), Convolutional Neural Networks (CNNs) (Unterkapitel 2.2) und GANs (Unterkapitel 2.3). Außerdem wird untersucht, welche Ansätze es bereits zur Generierung von synthetischen Daten mithilfe von GANs gibt (Unterkapitel 2.4). Abschließend werden in Kapitel 2 noch die für die vorliegende Studienarbeit verwendeten Beispieldaten betrachtet.

In Kapitel 3 werden GANs zur Erzeugung von synthetischen Trainings-Daten (Unterkapitel 3.1) und Klassifizierungs-KNNs zur Bewertung dieser Bilder (Unterkapitel 3.2) konzipiert. Es werden auch die Eigenschaften von KNNs beschrieben, auf die bei der Konzeption besonders geachtet werden muss (Unterkapitel 3.1). In Kapitel 4 wird die Umsetzung der in Kapitel 3 geschilderten Konzepte erläutert (Unterkapitel 4.2 bis 4.4). Es wird zudem beschrieben, welche Software-Werkzeuge und welche Frameworks für die Implementierungen verwendet werden (Unterkapitel 4.1).

In Kapitel 5 werden die Untersuchungsergebnisse der vorliegenden Studienarbeit vorgestellt. Es werden zunächst das Vorgehen für die Erhebung und die Bewertung der Untersuchungsergebnisse beschrieben (Unterkapitel 5.1). Anschließend werden die mit GANs erzeugten synthetischen Datensätze beschrieben und mit den echten Datensätzen verglichen (Unterkapitel 5.2). Letztlich werden die Testergebnisse der auf verschiedenen Datensätzen trainierten Klassifizierungs-KNNs verglichen. In Kapitel 6 werden die vorher aufgeführten Ergebnisse evaluiert. Kapitel 7 fasst die Erkenntnisse der vorliegenden Studienarbeit zusammen und gibt einen kurzen Ausblick in die Zukunft.

## **2. Stand der Technik**

In diesem Kapitel werden alle für das Verständnis der vorliegenden Studienarbeit relevanten Grundlagen erläutert. Es wird auch auf Best Practices und beachtenswerte Probleme hingewiesen. Zuerst wird in Kapitel 2.1 im Allgemeinen auf die Akteure und Strukturen sowie die dynamischen Abläufe in Bezug auf KNNs eingegangen. Anschließend wird in Kapitel 2.2 im Speziellen behandelt, wie mithilfe von KNNs Bilddaten klassifiziert werden können. Darauf aufbauend werden in Kapitel 2.3 die Grundlagen der GAN-Technologie beleuchtet. Auch hier werden die Akteure und Strukturen sowie die dynamischen Abläufe betrachtet. In Kapitel 2.4 wird darauf eingegangen, wie mit einem GAN ein umfangreicher synthetischer Trainings-Datensatz erzeugt werden kann. Abschließend werden in Kapitel 2.5 die in der vorliegenden Studienarbeit verwendeten Beispieldaten vorgestellt.

### **2.1. Künstliche Neuronale Netze**

Ein tierisches Lebewesen verarbeitet und speichert Informationen in seinem Gehirn. Die Gehirne von Lebewesen bestehen aus vielen vernetzten Recheneinheiten und arbeiten mit einem hohen Maß an Parallelität. Die komplexe Netz-Struktur ermöglicht es dem Gehirn, komplexe Problemstellungen wie z. B. Bilderkennung auch mit einem hohen Maß an Unschärfe in den Daten zu lösen. Die Struktur von Computern unterscheidet sich davon wesentlich. Daher sind Computer mit herkömmlichen Algorithmen auch nicht in der Lage, die gleichen Problemstellungen zu lösen wie das Gehirn [11, 12]. Mitte des 20. Jahrhunderts entwickelten Wissenschaftler erste mit Computern umsetzbare Modelle des menschlichen Gehirns, um diese Problemstellungen zu lösen [12]. Diese Modelle werden Künstliche Neuronale Netze (KNNs) genannt, da sie natürliche Neuronale Netz wie z. B. das „Gehirn“ künstlich modellieren.

Nachfolgend werden zentrale Aspekte der KNN-Technologie skizziert. Dazu wird ein statisches Modell der Akteure und ein dynamisches Modell der Abläufe aufgestellt.

### 2.1.1. Statisches Modell der Akteure

KNNs bestehen aus vielen Neuronen (Akteure), die über Verbindungen miteinander verknüpft sind (Struktur). Diese Verbindung leiten Signale durch veränderbare „Gewichte“ unterschiedlich stark von einer Einheit zur nächsten weiter. Durch die Veränderung dieser Gewichte lernt ein KNN, bestimmte Muster in Daten zu erkennen [12].

#### Die Struktur von KNNs

Typischerweise sind KNNs aus einer „Schicht“ von Neuronen für den eingehenden Informationsfluss (Eingabeschicht), einem Netzwerk aus mehreren Neuronen-Schichten für die Informationsverarbeitung (versteckte Schichten) und eine Schicht von Neuronen für den ausgehenden Informationsfluss (Ausgabeschicht) aufgebaut [2, 12]. Abbildung 2.1 stellt die beschriebene Struktur nochmal grafisch dar. Nachfolgend werden die drei Schicht-Typen von KNNs näher beschrieben:

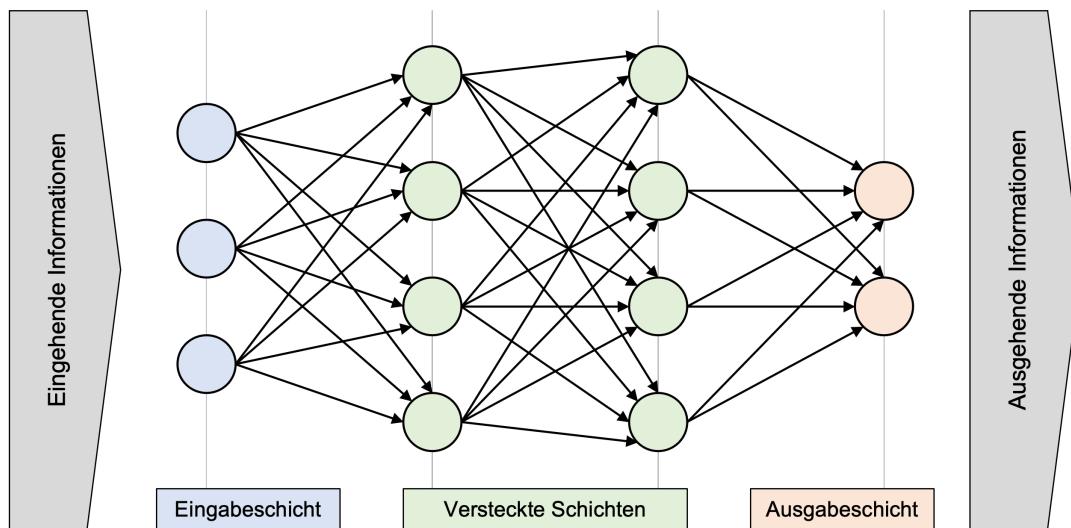


Abbildung 2.1.: Die Schichten-Struktur eines mehrschichtigen KNN:

Typischerweise bestehen KNNs aus einer einer Eingabeschicht, einem Netzwerk aus mehreren versteckten Schichten und einer Ausgabeschicht. In dem hier dargestellten vollverknüpften FeedForward-KNN leitet jedes Neuron einer Schicht  $i$  sein Berechnungsergebnis ausschließlich an alle Neuronen der darauf folgenden Schicht  $i + 1$  weiter.

**1. Eingabeschicht:**

Schichten diesen Typs formen komplexe Informationen wie z. B. Bild-, Text- oder Audiodaten um, sodass sie von den folgenden versteckten Schichten eines KNN verarbeitet werden können. Beispielsweise werden Bilddaten häufig in ihre Pixel zerlegt [2, 3, 12].

Zusammengehörige Teilbereiche eines Datensatzes (z. B. alle Pixel einer Bilddatei) werden genau wie bei natürlichen Neuronalen Netzen parallel in ein KNN eingelesen. Dafür verwendet die Eingabeschicht so viele künstliche Neuronen, wie ein Datensatz Teilbereiche hat. Diese künstlichen Neuronen nehmen die aufgespaltenen Informationen entgegen und reichen sie ohne weitere Berechnungen oder Modifikationen an die nächste Schicht des KNN weiter [2, 5].

**2. Versteckte Schichten:**

In den versteckten Schichten eines KNN findet der wesentliche Teil der Berechnungen statt, die zur Problemlösung führen. Die Neuronen der Eingabeschicht des KNN leiten die ausgehenden Informationen an die Neuronen der ersten versteckten Schicht eines KNN weiter (siehe Abbildung 2.1). Die Neuronen der versteckten Schichten führen Berechnungen aus und leiten anschließend die berechneten Ergebnisse an die nächste versteckte Schicht weiter. Die letzte versteckte Schicht leitet die verarbeiteten Informationen an die Ausgabeschicht weiter. KNNs können beliebig viele versteckte Schichten mit verschiedenen Neuronenzahlen haben [3, 5].

**3. Ausgabeschicht:**

Die Ausgabeschicht besteht typischerweise aus mehreren künstlichen Neuronen, die jeweils ein eigenes Ergebnis der Berechnungen des KNN repräsentieren. Die Ausgabe eines KNN muss je nach Problemstellung unterschiedlich interpretiert werden. Zum Beispiel errechnen Klassifizierungs-KNNs kein eindeutiges Ergebnis, sondern eine Verteilung der Klassenzugehörigkeits-Wahrscheinlichkeit [3, 5].

Ein KNN, das Bilder als Abbildungen entweder von Katzen oder Hunden klassifiziert, hat dementsprechend zwei künstliche Neuronen in der Ausgabeschicht und produziert für die Entscheidungen „Katze“ und „Hund“ jeweils eine Zugehörigkeits-Wahrscheinlichkeit. Die Klasse mit der höheren Zugehörigkeits-Wahrscheinlichkeit wird als Klassifizierungs-Entscheidung des KNN angenommen.

Häufig werden KNNs „vollverknüpft“ konstruiert. Das bedeutet, dass jedes Neuron einer Schicht  $i$  mit jedem Neuron der darauf folgenden Schicht  $i + 1$  verbunden ist (siehe Abbildung 2.1). Eng damit verbunden ist die in Abbildung 2.1 dargestellte „FeedForward-Architektur“. FeedForward-KNNs lassen Informationsfluss ausschließlich in Richtung der Ausgabeschicht zu. Die FeedForward-Architektur ist die Grundlage für andere KNN-Architekturen, bei denen Informationen z. B. auch zurück in Richtung Eingabeschicht fließen können [2, 12, 13].

In der Realität lassen sich mit nur zwei versteckten Schichten sehr beeindruckende Ergebnisse z. B. bei einfachen Bilderkennungsproblemen erzielen. Für komplexere Probleme und höhere Genauigkeit ist es jedoch häufig von Nöten, mehr versteckte Schichten in ein KNN einzubauen. Die Generierung von für Menschen verständlichen Texten bedarf für gute Ergebnisse bis zu 96 Schichten [14]. Besonders KNNs für Bilderkennungsprobleme profitieren von vielen versteckten Schichten. Im Jahr 2017 erzielte das KNN „SENet“ mit 154 Schichten einen Platz in den Top 5 der bekannten „ImageNet Object Localization Challenge“ für Bilderkennungsprobleme [15, 16]. Theoretisch können beliebig viele versteckte Schichten eingefügt werden. Praktisch steigt der Rechenaufwand jedoch stark mit der Anzahl der Schichten und Neuronen eines KNN. Daher begrenzt die eingesetzte Hardware die Anzahl von Schichten und Neuronen [3, 5, 13].

## Die Akteure von KNNs: Künstliche Neuronen

Aus den Ausführungen zu der Struktur von KNNs geht hervor, dass KNNs aus einer Menge künstlicher Neuronen ( $N$ ) bestehen, die über eine Menge von Verbindungen ( $V$ ) verknüpft sind. Um das Statische Modell eines KNNs vollständig verstehen zu können, werden nachfolgend die Akteure von KNNs – die künstlichen Neuronen – beschrieben.

Die Neuronen im menschlichen Gehirn sind biologische Recheneinheiten, die Informationen in Form von elektrischen Signalen verarbeiten. Neuronen im Gehirn können ihre synaptischen Verbindungen anpassen, um zu lernen. Die äquivalenten künstlichen Neuronen in KNNs sind mathematische Funktionen, die auf den eingehenden numerischen Informationsfluss angewendet werden. Damit ein KNN Muster erkennen lernt, werden Gewichte eingesetzt, die den ausgehenden Informationsfluss an jeder Verbindung  $v$  separat anpassen. Die Änderung der Gewichte findet auch hier basierend auf den Berechnun-

gen der Neuronen statt. Gewichte sind lediglich Zahlenwerte, die mit dem Ausgehenden Informationsfluss verrechnet werden [12].

Ein KNN wird als Tripel  $(N, V, w)$  dargestellt. Ein Neuron  $n_i \in N$  wird über seinen Index in der Schicht  $i$  lokalisiert. Eine Verbindung des Neurons aus der Schicht  $i$  mit einem Neuron aus der Schicht  $j$  wird als Tupel  $(i, j)$  geschrieben ( $V = \{(i, j) \mid i, j \in \mathbb{N}\}$ ). Einer Verbindung ist immer ein Gewicht zugeordnet ( $w : V \rightarrow \mathbb{R}$ ). Somit kann das Gewicht zur Verbindung  $(i, j)$  eindeutig als  $w_{i,j}$  dargestellt werden [12].

Die von den Neuronen durchgeführten Berechnungen lassen sich in die drei Bestandteile Propagierungsfunktion, Aktivierungsfunktion und Ausgabefunktion (siehe Abbildung 2.2) unterteilen [2, 12]. Diese werden nachfolgend genauer beschrieben.

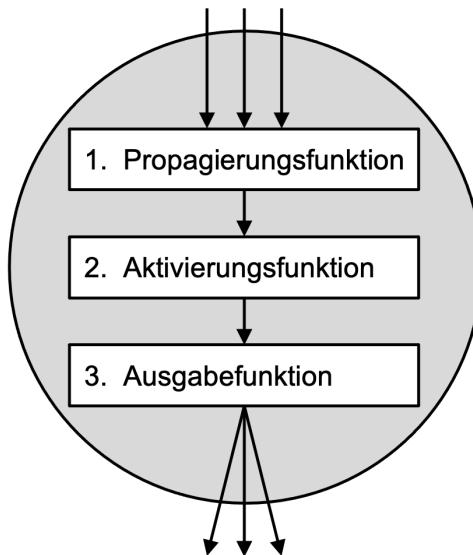


Abbildung 2.2.: Der Aufbau eines künstlichen Neurons:

Ein künstliches Neuron verarbeitet Daten über drei mathematische Funktionen. Die Propagierungsfunktion nimmt Daten von einem anderen Neuron entgegen und aggregiert diese. Auf die aggregierten Daten wird die Aktivierungsfunktion angewendet.

### 1. Propagierungsfunktion:

Die Propagierungsfunktion  $f_{prop}$  eines Neurons  $n_j$  dient dazu, die eingehenden Informationen von mehreren anderen Neuronen zu akkumulieren. Dazu werden die jeweiligen Gewichte  $w_{i,j}$  auf die Ausgaben der Neuronen  $n_i$  angewendet. Die gewichteten Ausgaben der Neuronen  $n_i$  werden anschließend aggregiert. Die verbreitetste Propagierungsfunkti-

on ist die gewichtete Summe:

$$f_{wSum} = \sum_{i \in I} (w_{i,j} \cdot out_i) .$$

Beispiele für weitere Propagierungsfunktionen sind der Maximalwert, der Minimalwert und auch das gewichtete Produkt. Das Ergebnis der Propagierungsfunktion des Neurons  $n_j$  heißt Nettoinput ( $net_j$ ) [12, 17, 18].

## 2. Aktivierungsfunktion:

Die Neuronen im Gehirn haben einen Aktivierungszustand  $a$ . Der Aktivierungszustand beschreibt, wie stark ein Neuron gereizt bzw. aktiviert ist. Bei KNNs erhöhen die aggregierten Signale aus der Propagierungsfunktion den Aktivierungszustand eines Neurons. Die Aktivierungsfunktion  $f_{akt}$  eines künstlichen Neurons ist nach dem biologischen Vorbild i. d. R. eine Funktion mit einem Schwellwert. Die Funktion und ihr Schwellwert beeinflussen die Ergebnisse eines KNN [12, 18].

Eine Aktivierungsfunktion muss stetig differenzierbar sein, damit sie für das Training von mehrschichtigen KNN-Modellen mit dem Backpropagation-Verfahren eingesetzt werden kann (siehe unten) [17, 18, 19]. Aktivierungsfunktionen sollten monoton sein und langfristig steile Gradienten besitzen, um das Training von KNNs effizienter zu gestalten (siehe unten) [3, 5, 18]. Weitere positive Eigenschaften von Aktivierungsfunktionen sind ein niedriger Implementierungsaufwand und ein geringer Berechnungsaufwand [12, 17]. Nachfolgend werden einige gängige Aktivierungsfunktionen für KNNs vorgestellt. In Abbildung 2.3 werden die Graphen dieser Aktivierungsfunktionen aufgeführt.<sup>1</sup>

- Lineare Funktionen sind die einfachste Form einer Aktivierungsfunktion. Sie können leicht implementiert und schnell berechnet werden. Allerdings eignen sie sich nur für einfache Problemstellungen und können ausschließlich in KNNs ohne verdeckte Schichten effektiv genutzt werden [5, 18].
- Mit der binären Stufenfunktion können komplexere Problemstellungen gelöst werden. Bis zu dem Schwellwert  $x_s$  gibt die binäre Stufenfunktion konstant den Wert  $y_{low}$  zurück. Mit dem Überschreiten des Schwellwerts  $x_s$  gibt die binäre Stufenfunktion konstant einen Wert  $y_{high}$  zurück. Der direkte Wechsel zwischen konstanten Werten macht die binäre Stufenfunktion am Schwellwert  $x_s$  nicht differenzierbar.

---

<sup>1</sup>Die multivariate SoftMax-Funktion wird in Abbildung 2.3 nicht gezeigt

Dadurch ist die binäre Stufenfunktion ungeeignet für das Trainieren mehrschichtiger KNNs. Die binäre Stufenfunktion kann einfach implementiert und schnell berechnet werden [12, 18].

- Sigmoide Funktionen haben einen S-förmigen Verlauf, der dem der binären Stufenfunktion ähnelt. Allerdings sind die sigmoiden Funktionen durchgängig differenzierbar und daher auch für mehrschichtige KNNs geeignet. Beispiele für solche sigmoiden Aktivierungsfunktionen sind die Sigmoid-Funktion und der Tangens Hyperbolicus. Sigmoide Funktionen konvergieren an den Rändern gegen konstante Werte, was ein Verschwinden des Gradienten bewirkt. Das beeinflusst das Training von KNNs oft negativ [12, 18].
- Die SoftMax-Funktion ist eine besondere Aktivierungsfunktion, da sie vorrangig für Klassifizierungsprobleme und dort i. d. R. nur auf der Ausgabeschicht verwendet wird. Das liegt daran, dass die SoftMax-Funktion nur mit  $k$  Variablen für  $k$  Klassen arbeitet. Die  $k$  reellen Eingabewerte werden auf Wahrscheinlichkeits-Werte im Intervall  $[0, 1]$  abgebildet, sodass die  $k$  Wahrscheinlichkeits-Werte sich zu 1 summieren. Aus einem beliebigen reellen Eingabe-Vektor entsteht eine Wahrscheinlichkeitsverteilung über die  $k$  Klassen [5, 17].
- Die Rectified Linear Unit (ReLU)-Funktion hat für  $x > 0$  den Verlauf der Identitätsfunktion  $f(x) = x$  und für  $x \leq 0$  einen konstanten Wert von 0. Die ReLU-Funktion ist am Schwellwert  $x = 0$  nicht stetig und daher eigentlich nicht durchgängig differenzierbar. Durch Festlegen eines Gradienten wird dieses Problem in der Praxis umgangen. Der Gradient der ReLU-Funktion hat für  $x > 0$  einen konstant guten, einfach zu berechnenden Wert, was den Trainingsprozess i. d. R. effizienter gestaltet als bei den sigmoiden Funktionen. Der Gradient der ReLU-Funktion verschwindet jedoch für  $x \leq 0$  [6, 20, 21].
- Die Leaky ReLU-Funktion modifiziert die ReLU-Funktion, sodass für  $x \leq 0$  anstatt des konstanten Nullwerts eine sehr flache lineare Funktion mit der Steigung  $a$  angenommen wird. Dadurch verschwindet der Gradient für  $x \leq 0$  nicht [5, 6].

### 3. Ausgabefunktion:

Die Ausgabefunktion  $f_{out}$  eines Neurons verschiebt das Ergebnis der Aktivierungsfunkti-

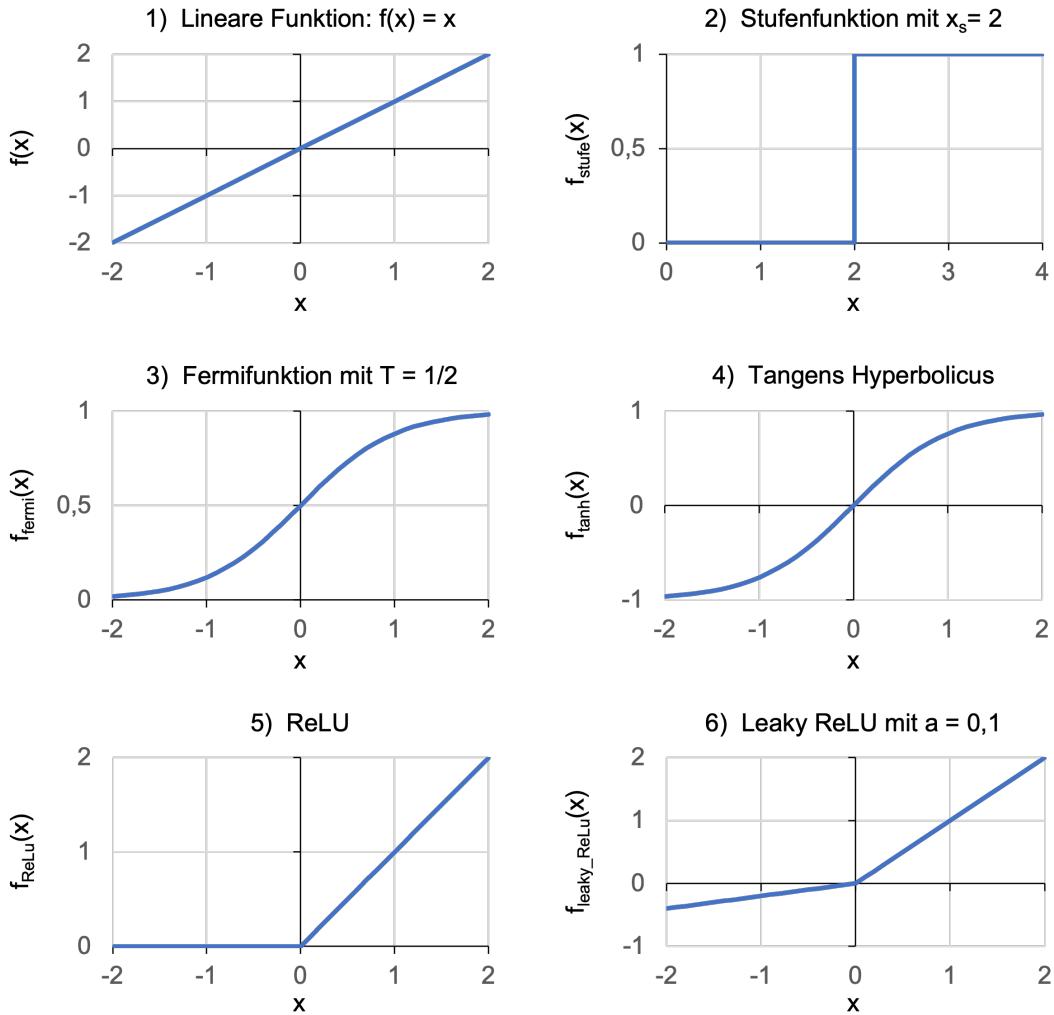


Abbildung 2.3.: Ein Überblick über gängige Aktivierungsfunktionen

on. Als Ausgabefunktion wird häufig die Identitätsfunktion genutzt, sodass das Ergebnis  $a_j$  der Aktivierungsfunktion auch das Ergebnis  $y_j$  der Ausgabefunktion ist [12, 18].

### 2.1.2. Dynamisches Modell der Abläufe

Basierend auf dem statischen Modell der Akteure wird nachfolgend das dynamische Modell der Abläufe von KNNs beschrieben. Es wird erläutert, wie mehrschichtige KNNs mit Fehlerminimierung und dem Backpropagation-Verfahren lernen, Problemstellungen

zu lösen. Dazu werden mehrschichtige, vollvernetzte KNNs mit einer reinen FeedForward-Architektur betrachtet.

„Lernen“ bezeichnet im Allgemeinen einen Prozess zur Anpassung an Umgebungsbedingungen. KNNs lernen hauptsächlich durch die Anpassung der Gewichte an den Verbindungen zwischen Neuronen. Beim Training von KNNs werden die Gewichte bei der Verarbeiten vieler tausend Beispieldatensätze iterativ angepasst. Dadurch lernen KNNs die für die Lösung eines Problems relevanten Merkmale zu erkennen. KNNs können diese Merkmale anschließend in ihnen unbekannten Datensätzen erkennen. Diese Fähigkeit wird Generalisierung genannt [5, 12, 18].

Die besagten Beispieldatensätze entsprechen in Format und Inhalt den Datensätzen, die dem KNN nach dem Training auch im praktischen Einsatz zugeführt werden. Damit KNNs die Fähigkeit zur Generalisierung erlangen, ist es wichtig, dass eine gewisse Menge und Verschiedenheit von Beispieldatensätzen vorliegt. Typischerweise bestehen solche Datensammlungen aus mehreren tausend Datensätzen, die möglichst gleichmäßig auf den Ergebnisbereich eines KNN verteilt sind [3, 22, 23]. Die Menge der Beispieldatensätze wird vor Beginn des Trainings in drei disjunkte Gruppen aufgeteilt [12, 24, 25]:

- Trainingsdaten:

Trainingsdatensätze werden im Lernprozess eingesetzt, um die Gewichte anzupassen. Der Anteil der Trainingsdaten an den Beispieldaten ist deutlich größer als der Anteil der Validierungs- und Testdaten.

- Validierungsdaten:

Mit den Validierungsdaten werden die Fortschritte während des Trainings eines KNN sichtbar gemacht. Auf Grundlage dieser Informationen können anschließend manuelle Struktur-Anpassungen (z. B. mehr Neuronen) an einem KNN vorgenommen werden. Bei der Validierung werden die Gewichte eines KNN nicht angepasst.

- Testdaten:

Mit den Testdaten werden die Ergebnisse der Trainings eines KNN nach dem Ende des Trainingsprozess überprüft. Wenn das KNN die Probleme in den Testdaten ausreichend genau lösen kann, war der Lernprozess erfolgreich.

Unter Umständen kann es dazu kommen, dass das Training eines KNN zufriedenstellend abläuft, aber das Testen fehlschlägt. Das kann durch verschiedene Effekte von un-

terschiedlichen Ursachen hervorgerufen werden. Ein häufig auftretendes Problem beim Training von KNNs ist das „Overfitting“. Beim Overfitting passt sich ein KNN zu stark an und spezialisiert sich auf die Trainingsdaten. Es findet also keine Generalisierung auf die Problemklasse statt, sondern eine Spezialisierung auf die Probleme in den Trainingsdaten [5]. Wenn die Vorhersagegenauigkeit auf den Trainingsdaten deutlich über der Vorhersagegenauigkeit auf den Validierungs- oder Testdaten liegt, besteht die Möglichkeit eines Overfittings.

KNNs können auf drei Arten trainiert werden. Jede Trainingsart ist nur für bestimmte Problembereiche und bestimmte Daten geeignet. Nachfolgend werden die drei Trainingsarten beschrieben:

1. Überwachtes Lernen:

Beim überwachten Lernen beinhalten die Beispieldatensätze nicht nur die Daten über das Problem (z. B. Pixel eines Bildes), sondern auch eine Angabe zum erwarteten Ergebnis, das das KNN produzieren soll. Dadurch ist im Voraus schon definiert, welche Klassen das KNN erkennen soll. Durch einen Soll-Ist-Vergleich kann das KNN seinen Fehler berechnen und mit diesem Fehler die Gewichte anpassen [12, 24].

2. Bestärkendes Lernen:

Beim bestärkenden Lernen wird dem KNN nach Berechnung des Ergebnisses für einen Trainingsdatensatz in Form einer Zahl oder eines Wahrheitswerts mitgeteilt, ob und wie genau das errechnete Ergebnis mit dem erwarteten Ergebnis übereinstimmt. In diesem Fall ist nicht vorher festgelegt, welche Muster-Klassen ein KNN erkennen soll. Durch die Rückmeldungen wird das KNN jedoch in bestimmte Richtungen gelenkt [12, 24].

3. Unüberwachtes Lernen:

Beim unüberwachten Lernen erhält ein KNN keinerlei Rückmeldung zu dem errechneten Ergebnis. In diesem Fall muss das KNN alleine Muster-Klassen bilden. Diese Form des Lernens ist dem Lernprozess im Gehirn am ähnlichsten [12, 24].

In der vorliegenden Studienarbeit werden Klassifizierungsprobleme auf Bilddaten betrachtet. Diese Probleme fallen in das Gebiet des überwachten Lernens. Daher wird weiterführend nur noch das überwachte Lernen betrachtet.

Der „Loss“ genannte Vorhersage-Fehler eines KNN ist das zentrale Element beim Trainieren von KNNs mit überwachtem Lernen. Der Loss kann auf verschiedene Arten berechnet werden, stellt aber immer die Abweichung zwischen einem vorgegebenen Ergebnis  $t$  und dem durch das KNN vorhergesagten Ergebnis  $\vec{y}$  dar. Das besagte vorgegebene Ergebnis ist nicht Teil der Eingabe für das KNN. Lediglich die zu klassifizierenden komplexen Informationen (z. B. Bilddateien) werden in ein KNN eingespeist [3, 24].

Die Ergebnisse  $t$  und  $\vec{y}$  sind bei Klassifizierungsproblemen i. d. R. Vektoren mit einem Wert für jede der Klassen des Problems. Für jede Klasse wird eine Zugehörigkeitswahrscheinlichkeit angegeben [3, 12]. Nachfolgend wird daher die Vektor-Schreibweise für  $\vec{t}$  und  $\vec{y}$  benutzt.

### **Loss und Loss-Funktionen**

Der Loss-Wert  $Err_p$  für einen Datensatz  $p$  errechnet sich aus der Abweichung des errechneten vom erwarteten Ergebnis und wird ebenfalls als Vektor dargestellt. Wie die Abweichung errechnet wird, bestimmt die gewählte Loss-Funktion  $f_{Err}$ . Loss-Funktionen errechnen dabei immer einen Loss-Wert, der alle Komponenten der Ergebnis-Vektoren einbezieht. Es gibt viele Loss-Funktionen, die für unterschiedliche Problemstellungen gute Ergebnisse liefern [6, 12]. Nachfolgend werden zwei Loss-Funktionen vorgestellt, die relevant für diese Studienarbeit sind:

1. Mean Squared Error (MSE):

Diese Loss-Funktion bildet einen Ergebnisdifferenz-Vektor  $\vec{e} = \vec{t} - \vec{y}$ . Die Summe der quadrierten Komponenten dieses Vektors geteilt durch die Anzahl der Komponenten ist der MSE-Loss-Wert. Eine gängige Abwandlung dieser Loss-Funktion ist der euklidische Abstand. Der MSE-Loss kann für viele Aufgabentypen eingesetzt werden. Besonders bei Regressionsaufgaben ist diese Loss-funktion geeignet. Die Funktion zur Berechnung des MSE-Loss sieht wie folgt aus [6, 12, 25, 26, 27]:

$$f_{Err}(\vec{t}, \vec{y}) = \frac{1}{m} \sum_{i=1}^m (t_i - y_i)^2.$$

## 2. Kreuzentropie:

Die Kreuzentropie wird vornehmlich für Klassifizierungsprobleme eingesetzt [6, 25].

Bei Klassifizierungsproblemen kann ein Datensatz genau einer Klasse richtig zugeordnet werden. Das erwartete Ergebnis  $\vec{t}$  ist daher ein Einheitsvektor. Der Term für die Berechnung der Kreuzentropie für den Anwendungsfall der Klassifizierung sieht wie folgt aus [5, 27, 28]:

$$f_{Err}(\vec{e}_p) = -\log(y_i) \text{ mit } t_i = 1.$$

KNN werden i. d. R. mehrfach mit den denselben Daten trainiert und anschließend mit den denselben Daten validiert. Ein solcher Durchlauf wird „Epoche“ genannt. Über die Loss-Werte für alle Validierungsdatensätze einer Epoche kann ein Durchschnitt gebildet werden. Ist dieser durchschnittliche Loss-Wert niedrig, ist ein KNN gut trainiert. Trägt man die durchschnittlichen Loss-Werte (Y-Achse) für mehrere Epochen (X-Achse) in ein Koordinatensystem ein, ergibt sich eine Lernkurve. Diese Lernkurve ist im Idealfall streng monoton fallend und konvergiert schnell gegen Null [3, 12, 24].

## **Das Gradientenabstiegs-Verfahren**

Das Ziel beim Training von KNNs ist es, den Loss-Wert zu senken und dadurch eine möglichst hohe Vorhersage-Genauigkeit zu erzielen. Um dieses Ziel zu erreichen, wird das Gradientenabstiegs-Verfahren eingesetzt. Mit dem Gradientenabstiegs-Verfahren können Minima einer Funktion bestimmt werden. Dabei wird von einem beliebigen Ausgangspunkt aus der Gradient der Loss-Funktion gebildet. Dieser Gradient gibt die Richtung der stärksten Änderung des Loss-Werts von dem Ausgangspunkt aus an. Um den Loss-Wert zu reduzieren, wird auf der Loss-Funktion ein Schritt in die durch den Gradienten vorgegebene Richtung gemacht. Dabei ist die Schrittweite proportional zur Steigung des Gradienten. Steile Gradienten trainieren ein KNN stärker als flache Gradienten. Der erreichte Punkt dient als neuer Ausgangspunkt. Um ein Minimum der Loss-Funktion zu finden, wird dieses Vorgehen wiederholt, bis der Gradient den Wert 0 annimmt [3, 12].

Der Gradientenabstieg findet i. d. R. einen Punkt auf der Loss-Funktion bei dem der Gradient den Wert 0 hat. Wie effizient und effektiv dies geschieht, ist von dem ersten Ausgangspunkt auf der Loss-Funktion und der Schrittweite abhängig. Unpassende Kon-

figurationen können zu Problemen führen. Nachfolgend werden drei solcher Probleme des Gradientenabstiegs beschrieben:

1. Lokale Minima:

Mit dem Gradientenabstieg wird nicht garantiert ein globales Minimum der Loss-Funktion gefunden. Im Gegenteil werden häufig lokale Minima gefunden, die unzureichend für das Training eines KNN sind. Es ist nicht erkennbar, ob das gefundene Minimum das globale Minimum der Loss-Funktion ist. Daher werden der erste Ausgangspunkt und die Schrittweite in der Praxis solange variiert, bis ein Minimum mit ausreichender Qualität gefunden wurde [3, 12, 17].

2. Erliegen des Trainings:

Die Steigung der Gradienten beeinflusst die Schrittweite des Gradientenabstiegs. Daher kann es zu starken Verzögerungen oder sogar zum Erliegen des Trainings kommen, wenn die Steigung des Gradienten Werte nahe 0 annimmt. Ein solches Verhalten tritt bei Minima, aber auch bei Sattelpunkten auf. Ein Sattelpunkt ist kein erwünschtes Ergebnis [3, 12, 17].

3. Überspringen tiefer Minima:

Es ist möglich, dass die Loss-Funktion Minima besitzt, die schmal (kleine Differenz auf der X-Achse), aber tief (große Differenz auf der Y-Achse) sind. Wenn die Steigung des Gradienten am Anfang eines solchen Minimums groß genug ist, kann das Minimum übersprungen werden. Das kann bedeuten, dass zwischen beiden Anfängen des Minimums hin und her gependelt oder ein anderes Minimum gefunden wird [3, 12, 17].

## Das Backpropagation-Verfahren

Das Backpropagation-Verfahren konkretisiert das oben beschriebene Gradientenabstiegs-Verfahren für den praktischen Einsatz mit mehrschichtigen KNNs. Über eine Lern-Regel definiert das Backpropagation-Verfahren, wie die Gewichte zwischen zwei beliebigen Neuronen eines mehrschichtigen KNN angepasst werden. Diese Lern-Regel fußt auf den Loss-Werten für die jeweiligen Neuronen. Es gibt allerdings nur für die Neuronen der Ausgabeschicht ein erwartetes Ergebnis  $\vec{t}$ . Für die Berechnung der Loss-Werte der Neuronen auf einer inneren Schicht  $i$  werden die Loss-Werte der Neuronen der nachfolgenden Schicht

$i+1$  zurück propagiert. Um die Gewichte einer KNN anzupassen, muss das KNN für jeden Datensatz vorwärts und rückwärts durchlaufen werden [5, 12].

Die Anpassung des Gewichts  $w_{i,j}$  an der Verbindung von einem beliebigen Neuronen  $n_i$  zu einem beliebigen Neuron  $n_j$  wird durch den Term

$$\Delta w_{i,j} = \eta \cdot y_i \cdot \delta_j$$

mit der Lern-Regel des Backpropagation-Verfahrens beschrieben. Dabei errechnet sich die Gewichtsänderung  $\Delta w_{i,j}$  aus dem Produkt der Ausgabe des Neurons  $n_i$  ( $y_i$ ) und dem Loss-Wert des Neurons  $n_j$  ( $\delta_j$ ) sowie einem „Lernrate“ genannten Proportionalitätsfaktor  $\eta$ . Die Ausgabe-Werte  $y$  werden für alle Neuronen des KNN bei einem Vorwärtsdurchlauf berechnet und für die Rückwärtige Propagierung vorgehalten. Für den Loss-Wert  $\delta_j$  müssen die folgenden zwei Fälle unterschieden werden [12, 29]:

1. Schicht  $j$  ist die Ausgabeschicht:

Der Loss-Wert  $\delta_j$  des Neurons  $n_j$  wird aus dem Gradienten  $\nabla Err$  der Loss-Funktion errechnet. Durch Ableitung der Fehlerfunktion nach dem Gewicht  $w_{i,j}$  ergibt sich

$$\delta_j = \nabla Err = f'_{Akt}(net_j) \cdot f'_{Err}(t, y_j).$$

Dabei ist  $f_{Akt}(net_j)$  die Aktivierungsfunktion des Neurons  $n_j$  und  $f_{Err}(t, y_j)$  die verwendete Loss-Funktion [12, 29].

2. Schicht  $j$  ist eine innere Schicht:

Die Ableitung der Loss-Funktion  $f'_{Err}(t, y_j)$  kann hier nicht für die Berechnung von  $\delta_k$  verwendet werden, da sie ein erwartetes Ergebnis  $t$  benötigt. Stattdessen werden die Loss-Werte  $\delta_k$  aller Neuronen der Schicht  $k = j + 1$  verwendet, die mit dem Neuron  $n_j$  verbunden sind. Dabei werden nur die durch die Gewichte  $w_{j,k}$  beeinflussten Teile des Loss-Werts  $\delta_k$  einbezogen. Für innere Schichten gilt [12, 29]

$$\delta_j = \nabla Err = f'_{Akt}(net_j) \cdot \sum_{k \in N_k} (\delta_k \cdot w_{j,k}).$$

Durch Zusammenführen der beiden Fälle ergibt sich die folgende Lernregel für mehrschichtige KNNs mit dem Backpropagation-Verfahren [12, 29]:

$$\Delta w_{i,j} = \eta \cdot y_i \cdot \delta_j \quad \text{mit}$$

$$\delta_j = \begin{cases} f'_{akt}(net_j) \cdot f'_{err}(y_j), & j \in \text{Ausgabeschicht} \\ f'_{akt}(net_j) \cdot \sum_{k \in N_k} (\delta_k \cdot w_{j,k}), & j \in \text{innere Schichten} \end{cases}.$$

Der erste Faktor der Lern-Regel des Backpropagation-Verfahrens ist die Lernrate. Die Lernrate kann konstant für den gesamten Trainingsprozess festgelegt werden. Das führt allerdings häufig zu den beschriebenen Problemen des Gradientenabstiegs-Verfahrens. Besser ist es, die Lernrate während des Trainings anzupassen. Dabei werden i. d. R. zuerst große Lernraten verwendet, um schnell grobe Lernfortschritte zu erzielen. Danach werden mit kleinen Lernraten präzisere Anpassungen vorgenommen. Außerdem können unterschiedliche Lernraten für unterschiedliche Neuronen-Schichten und sogar je Neuron festgelegt werden. Dabei werden i. d. R. für die Schichten nahe der Eingabeschicht größere Lernraten verwendet, als für Schichten nahe der Ausgabeschicht [5, 12].

Neben einer adaptiven Lernrate können die Probleme des Gradientenabstiegs-Verfahrens auch durch simulieren des Trägheitseffekts abgemildert werden. Angenommen die funktionale Abhängigkeit des Loss-Werts *Err* (*y*-Achse) von dem Wert eines Gewichts  $w_{i,j}$  (*x*-Achse) ist eine Landschaft mit Bergen und Tälern. Das Ziel des Gradientenabstiegs-Verfahrens ist es eine Kugel von einem beliebigen Punkt aus in das tiefste Tal der Landschaft zu befördern. Mit dem einfachen Gradientenabstiegs-Verfahren bleibt die Kugel vom einen auf den anderen Moment auf einem Sattelpunkt stehen und kommt aus flachen Tälern nicht mehr heraus (siehe Abbildung 2.4). Das reduziert die Wahrscheinlichkeit, dass die Kugel in dem tiefsten Tal landet. Durch den Trägheitseffekt kann die Kugel ihre Bewegungsenergie teilweise beibehalten und auf Plateaus eine gewisse Entfernung zurücklegen oder sogar Berg-Anstiege überwinden (siehe Abbildung 2.4) [3, 5, 12].

Um den Trägheitseffekt in dem Gradientenabstiegs-Verfahren zu simulieren, wird die Gleichung für  $\Delta w_{i,j}$  erweitert. Die erweiterte Lern-Regel bezieht die Gewichtsänderung im vorangegangenen Durchlauf mit Beispieldatensatz  $p_{n-1}$  in die Gewichtsänderung im aktuellen Durchlauf mit Beispieldatensatz  $p_n$  mit ein:

$$\Delta w_{i,j}(p_n) = \eta \cdot y_i \cdot \delta_j + \alpha \cdot \Delta w_{i,j}(p_{n-1}).$$

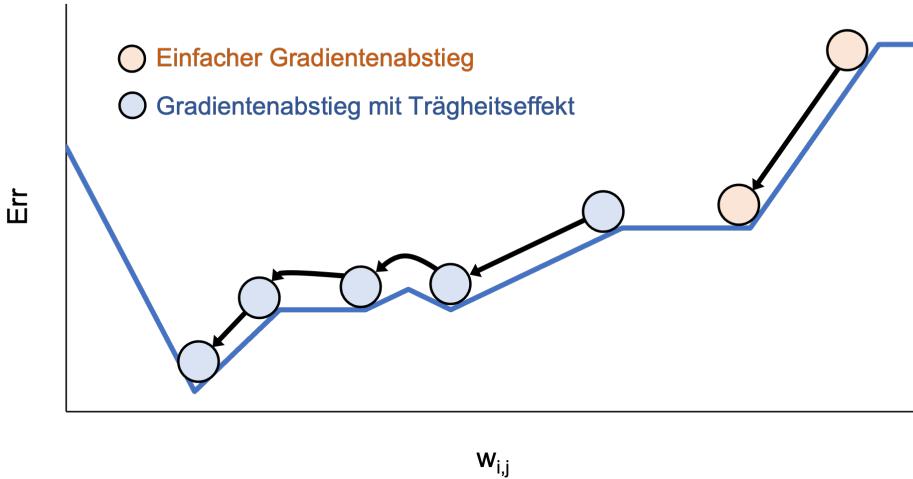


Abbildung 2.4.: Das Gradientenabstiegs-Verfahren mit integrierten Trägheitseffekt:

Die funktionale Abhängigkeit des Loss-Werts  $Err$  (y-Achse) von dem Wert eines Gewichts  $w_{i,j}$  (x-Achse) kann als Landschaft mit Bergen und Tälern interpretiert werden. Das Ziel des Gradientenabstiegs-Verfahrens ist es eine Kugel in das tiefste Tal der Landschaft zu befördern. Mit dem klassischen Gradientenabstiegs-Verfahren (in orange) können Plateaus und lokale Minima nicht überwunden werden. Durch einen integrierten Trägheitseffekt kann die Kugel ihre Bewegungsenergie auch bei Gradienten mit einer Steigung von 0 teilweise beibehalten und auf Plateaus eine gewisse Entfernung zurücklegen oder sogar Berg-Anstiege überwinden.

Der Faktor  $\alpha$  ist ein Koeffizient, mit dem der Trägheitseffekt moduliert werden kann. Üblicherweise wird  $\alpha$  zwischen 0,6 und 0,9 gewählt. Wird dieser Faktor zu groß gewählt, können auch tiefe Minima wieder verlassen werden [12, 17].

Das Root Mean Square Propagation (RMSprop) und das Adaptive Moment Estimation (ADAM) Verfahren sind optimierte Varianten des einfachen Gradientenabstiegs-Verfahrens. Beide Verfahren implementieren eigene Lernraten für jedes Neuron und passen diese Lernraten dynamisch an. Das ADAM-Verfahren nutzt zusätzlich das Momentum-Konzept. Beide Verfahren sind trotz ihres hohen Optimierungsgrads einfach einsetzbar und erzielen bei vielen Problem-Typen gute Ergebnisse [5, 17, 30].

Die Anpassung der Gewichte eines KNN ist eine iterativer, additiver Prozess. Durch Subtraktion von  $\Delta w$  von dem aktuellen Wert des Gewichts  $w_{alt}$  ergibt sich der neue Wert des Gewichts  $w_{neu} = w_{alt} - \Delta w$  [3]. Daher müssen die Gewichte eines KNN vor dem

Training initialisiert werden. Das Vorgehen bei der Initialisierung der Gewichte eines KNN kann den Trainingsprozess und die Leistungsfähigkeit eines KNN stark beeinflussen. Werden Gewichte mit dem Wert 0 initialisiert, können diese Gewichte nicht trainiert werden. Wenn alle Gewichte mit dem gleichen Wert initialisiert werden, passen sich alle Gewichte gleich an. Ein verbreitetes Vorgehen ist die Initialisierung mit Zufallswerten in Wahrscheinlichkeitsverteilungen in einem Intervall von z. B.  $[-0,5; 0,5]$  [12, 31].

## 2.2. Klassifizierung von Bilddaten (CNNs)

Mit KNNs können Problemstellungen aus verschiedenen Aufgabenfeldern gelöst werden. Allgemein bekannte Aufgabenfelder von KNNs sind die Assoziation, die Prognosbildung, die Segmentierung und die Klassifizierung [2, 17, 32] :

### 1. Assoziation:

Unter einer Assoziation versteht man im Allgemeinen das Herstellen einer Korrelation zwischen Objekten. Diese Korrelationen können auf beliebigen Parametern beruhen. Beispielsweise existiert eine Korrelation zwischen zwei Artikeln, die in einem Online-Shop häufig zusammen gekauft werden. Auch das menschliche Gedächtnis arbeitet mit Assoziationen. Dabei werden Erlebnisse an bestimmte Erinnerungen gebunden [17, 32].

### 2. Prognosbildung:

Prognosen basieren auf dem Herstellen von Assoziationen und sind daher eigentlich nur ein untergeordnetes Aufgabenfeld der Assoziation. In diesem Fall muss ein KNN einen funktionalen Zusammenhang modellieren. Dazu werden i. d. R. historische und aktuelle Daten eingesetzt, um das zukünftige Geschehen vorherzusehen. Ein Beispiel dafür ist „Predictive Maintenance“, bei dem vorhergesehen wird, wann eine Maschine gewartet werden muss [2, 17, 33].

### 3. Segmentierung:

Problemstellungen aus dem Aufgabenfeld der Segmentierung haben das Ziel, eine Gesamtmenge in klar getrennte Teilmengen aufzuteilen. Dabei wird jedes Element der Gesamtmenge genau einer Teilmenge zugeordnet. KNNs lösen Problemstellungen aus dem Aufgabenfeld der Segmentierung mittels unüberwachtem Lernen. Aus

diesem Grund müssen KNNs bei Segmentierungsaufgaben selbst erlernen, welche Untergruppen es gibt. Ein Beispiel für eine Segmentierungsaufgabe ist das Aufteilen eines Bildes anhand der in den Bereichen dargestellten Objekte [17, 34].

#### 4. Klassifizierung:

Klassifizierungsaufgaben ähneln den Segmentierungsaufgaben. Auch hier ist es das Ziel, die Objekte einer Gesamtmenge in klar differenzierte Teilmengen aufzuteilen. Im Unterschied zu Segmentierungsaufgaben lösen KNNs Klassifizierungsaufgaben mittels überwachtem Lernen. Daher kennt ein KNN die Teilemengen, in die eine Gesamtmenge unterteilt werden soll. Ein Beispiel für eine Klassifizierungsaufgabe ist das Einteilen von Kreditnehmern in Kreditwürdigkeits-Klassen basierend auf ihren personenbezogenen Daten und Finanzen [2, 17].

KNNs können Problemstellungen auf Daten verschiedener Datentypen lösen. Weit verbreitet sind Bild-, Ton- und Text-Daten [1]. Die vorliegende Studienarbeit beschränkt sich auf Klassifizierungsaufgaben und dabei besonders auf Bilderkennungsprobleme. Die in dieser Studienarbeit betrachteten Bilddaten sind Fotografien. Diese Fotografien zeigen bei einer Klassifizierungsaufgabe ein (1) Objekt, das klassifiziert werden soll.

Damit ein KNN eine Bilddatei verarbeiten kann, wird das Bild in seine Pixel aufgespalten. Dabei ist zu beachten, dass ein Pixel in einem Schwarz-Weiß-Bild i. d. R. nur aus einem Wert von 0 – 255 besteht, während bei einem Farb-Bild typischerweise drei Werte zwischen 0 und 255 (Rot, Grün, Blau) je Pixel bestehen [17, 26, 35].

Wie bereits im Kapitel zu KNNs beschrieben, werden für ein effizientes Training von KNNs mit einem Gradientenabstiegs-Verfahren langfristig möglichst steile Gradienten benötigt. Solche steilen Gradienten treten bei vielen Aktivierungsfunktionen in der Nähe des Nullpunkts auf (siehe Abbildung 2.3). Das ist ein Grund dafür, dass die Werte der einzelnen Pixel häufig von dem Intervall [0, 255] auf ein Intervall wie z. B. (-1, 1) überführt werden (siehe Abbildung 2.5).

Mit konventionellen KNNs können beeindruckende Ergebnisse bei einfachen Bilderkennungsaufgaben erzielt werden. In der praktischen Anwendung sind klassische KNNs jedoch selten geeignet. Denn für jeden betrachteten Pixel eines Bilds benötigt ein klassisches KNN ein Neuron in der Eingabeschicht. Klassische vollverknüpfte KNNs (siehe Kapitel 2.1) leiten alle Signale einer Schicht  $i$  in jedes Neuron der darauf folgenden

Schicht  $i + 1$  weiter. Die Anzahl der Gewichte zwischen beiden Schichten ergibt sich daher aus dem Produkt der Anzahl ihrer Neuronen. In der Praxis müssen Bilder mit vielen Tausend Pixeln betrachtet werden. Das Training klassischer KNNs mit einer so hohen Anzahl von Neuronen und Gewichten ist nicht praktikabel [3, 6, 17].

## Convolutional Neural Networks

In den letzten Jahren ist ein anderer Ansatz für KNNs zur Bilderkennung in den Fokus gerückt, der das Problem der Größe von KNNs angeht. „Convolutional Neural Networks“ verdichten die eingehenden Daten zu weniger, aber komplexeren Informationen und benötigen daher weniger Gewichte und Neuronen als klassische KNNs. Damit diese Verdichtung zu Informationen funktioniert, müssen die zu Grunde liegenden Daten die Eigenschaft aufweisen, dass nebeneinander liegende Datenpunkte semantisch in Beziehung stehen [5, 6, 17].

Bilddaten sind das populärste Beispiel für solche Daten, die mit CNNs verarbeitet werden können. Bei Bilddaten sind nebeneinanderliegende Pixel stark semantisch verbunden. Das erkennt man daran, dass das Tauschen von Pixel-Reihen oder -Spalten den Inhalt eines Bilds verändern. CNNs nutzen diese Eigenschaft aus, um Merkmale von Objekten in Bildern wie z. B. Kanten zu erkennen, die sich über größere Bereiche erstrecken. Über mehrere Schichten hinweg werden einfache Merkmale wie Kanten zu komplexeren Merkmalen wie z. B. Nasen oder Augen zusammengefügt. Mit diesem „Merkmals-Lernen“ können Neuronale Netze in vielen Bereichen nicht nur effizienter trainiert werden, sondern auch noch höhere Vorhersage-Genauigkeiten als klassische KNNs bei Bilderkennungsaufgaben erzielen [3, 5, 6, 17].

CNNs behalten die Grundstruktur eines klassischen KNN aus Eingabeschicht, versteckten Schichten und Ausgabeschicht bei. Zudem werden die Schichten eines CNN in Schichten für das Merkmals-Lernen und Schichten für das Klassifizieren unterteilt. Ein CNN besitzt relativ viele Schichten für das Merkmals-Lernen und darauf folgend relativ wenige klassische KNN-Schichten für das Klassifizieren [5, 17].

Die Eingabeschicht von CNNs unterscheidet sich wesentlich von der Eingabeschicht von KNNs. Das liegt daran, dass Daten nicht in Form einer eindimensionalen Liste wie bei einem klassischen KNN eingelesen werden, sondern als mehrdimensionale Struktur. Bei

einer Bilddatei z. B. handelt es sich i. d. R. um eine dreidimensionale Struktur mit einer Höhe, einer Breite und einer Tiefe (Anzahl Farbkanäle: Rot, Grün, Blau). In den Schichten für das Merkmals-Lernen werden die Daten in eine Form gebracht, die von klassischen KNN-Schichten verarbeitet werden kann [5, 17].

Für die folgenden Erläuterungen, werden der Einfachheit halber Schwarz-Weiß-Bilddaten mit nur einem Farbkanal betrachtet (siehe linke Matrix in Abbildung 2.5). Dadurch können die Bilddaten als zweidimensionales Feld und nicht als dreidimensionaler Quader dargestellt werden. In diesem Feld werden die Werte der Pixel von dem Intervall  $[0, 255]$  (siehe mittlere Matrix in Abbildung 2.5) auf das Intervall  $(-1, 1)$  skaliert abgetragen (siehe rechte Matrix in Abbildung 2.5).

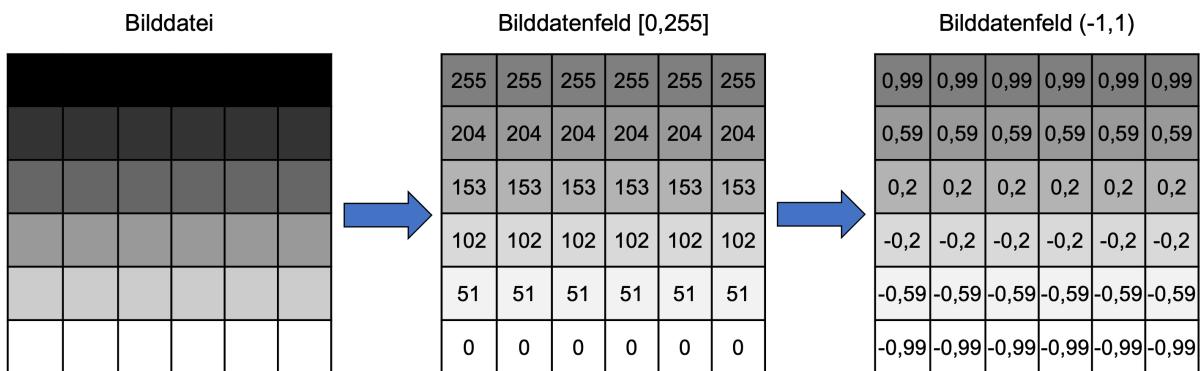


Abbildung 2.5.: Bilddaten für die Verarbeitung mit KNNs formatieren:

Jede der drei Matrizen ist eine Repräsentation der selben Bilddatei. Auf der linken Seite werden die Werte der Pixel mittels Farbe kenntlich gemacht. In der Mitte werden die Werte der Pixel mit Zahlen im Intervall  $[0, 255]$  gezeigt. Auf der rechten Seite werden die Werte der Pixel von dem Intervall  $[0, 255]$  auf das Intervall  $(-1, 1)$  übertragen.

Die versteckten Schichten für das Merkmals-Lernen bestehen i. d. R. aus „Faltungsschichten“, Aktivierungsschichten, Batch-Normalisierungs-Schichten und „Bündelungsschichten“ [5, 17, 34]. Nachfolgend werden diese vier Schicht-Arten beschrieben:

### 1. Faltungsschicht:

Faltungsschichten sind für die Filterung von Daten zuständig. Dabei werden verschiedene Filtermuster separat auf ein Bild angewendet. Ein Filtermuster ist ein Datenfeld mit den gleichen Dimensionen wie die zu filternden Daten und repräsentiert ein Merkmal, das in den Daten erkannt werden soll. Die Filtermuster haben die gleiche Tiefe wie das zu

filternde Datenfeld. Im Beispiel aus Abbildung 2.6 haben das zu filternde Datenfeld und das Filtermuster jeweils zwei Dimensionen (Tiefe = 1). Filtermuster sind quadratisch und haben eine Seitenlänge, die mindestens zwei, aber kleiner als die des zu filternden Datenfelds ist. Meistens werden mit kleinen Filtermustern bessere Ergebnisse als mit großen Filtermustern erzielt. Die Filtermuster beinhalten die Gewichte, die für das Training eines CNN angepasst werden müssen. Diese Gewichte sind für jedes Filtermuster unterschiedlich [5, 24, 34].

Die Filtermuster werden separat auf das zu filternde Datenfeld angewendet, um jeweils eine „Merkmalskarte“ (Feature-Map) zu erzeugen. Merkmalskarten sind Datenfelder, die über die abgetragenen Zahlenwerte zeigen, wo ein Merkmal in dem zu filternden Datenfeld auftritt. Merkmalskarten haben die gleichen Dimensionen wie das zu filternde Datenfeld. Allerdings ist die Tiefe der Merkmalskarte i. d. R. kleiner als die Tiefe des zu filternden Datenfelds. Um eine Merkmalskarte zu erzeugen, wird ein Filtermuster Schritt für Schritt an dem zu filternden Datenfeld angelegt und das Skalarprodukt der überschneidenden Bestandteile von Filtermuster und zu filterndem Datenfeld gebildet (siehe Abbildung 2.6). Die Schrittweite (Stride) kann dabei theoretisch frei gewählt werden. In der Praxis sind kleine Schrittweiten von eins oder zwei verbreitet [5, 34]. Im Beispiel aus Abbildung 2.6 wird eine Schrittweite von 1 genutzt.

Es gibt verschiedene Ansätze, wie Filtermuster an das zu filternde Datenfeld angelegt werden dürfen. Ein Ansatz ist, dass kein Teil des Filtermusters den Rand des zu filternden Datenfelds überschreiten darf. Dadurch verkleinert sich die Merkmalskarte (Breite und Höhe) im Vergleich zu dem zu filternden Datenfeld. Das ist meist nicht wünschenswert, da dadurch relevante Informationen verloren gehen können [5, 34]. Ein anderer Ansatz ist, dass ein Filtermuster mit dem zentralen Datenpunkt an dem zu filternden Datenfeld anliegen muss (siehe Abbildung 2.6). Eine Voraussetzung dafür ist, dass die Filtermuster eine ungerade Seitenlänge haben. Bei diesem Ansatz steht ein Teil des Filtermusters über den Rand des zu filternden Datenfeldes hinaus. Dadurch ist einigen Datenpunkten des Filtermusters kein entsprechender Datenpunkt auf dem zu filternden Datenfeld zugeordnet.

Um das Filtern trotzdem ordnungsgemäß ausführen zu können, wird das „Padding“ eingesetzt. Bei diesem Verfahren werden an den Rändern die notwendigen Zeilen und Spalten symmetrisch angefügt. Diese Zeilen und Spalten können mit unterschiedlichen

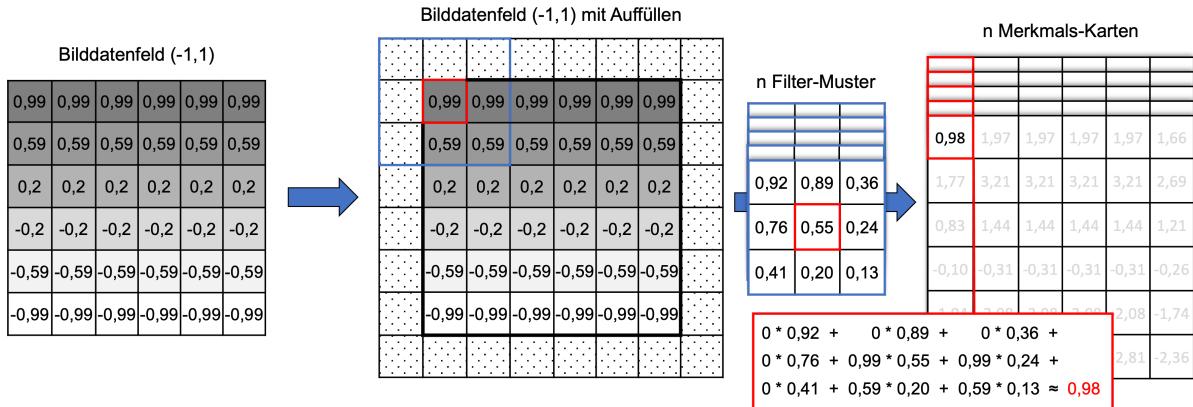


Abbildung 2.6.: Der Filterungsprozess einer Faltungsschicht:

Die vorbereiteten Bilddaten aus Abbildung 2.5 werden mit einem Zero-Padding an jeder Seite um einen Pixel erweitert. Die Filtermuster einer Faltungsschicht werden separat mit dem zentralen Datenpunkt an das Bilddatenfeld (Padding ausgenommen) angelegt. Über das Skalarprodukt der sich überschneidenden Datenpunkte des Filtermusters und des Bilddatenfelds, wird ein Datenpunkt in der resultierenden Merkmalskarte errechnet. Mit der Schrittweite 1 werden die Filtermuster von der oberen linken Ecke des Bilddatenfelds über das Bilddatenfeld bewegt, bis der zentrale Datenpunkt der Filtermusters an dem Datenpunkt in der rechten unteren Ecke des Bilddatenfelds anliegt.

Werten befüllt werden. Eine verbreitete Variante ist das Zero-Padding, bei dem ausschließlich Null-Werte verwendet werden [5, 17, 24]. Abbildung 2.6 verdeutlicht diesen Ansatz zum Erzeugen einer Merkmalskarte.

Für die Implementierung einer Faltungsschicht ist es oft hilfreich, die Ausgabegröße nach der Faltung berechnen zu können. Mithilfe der folgenden Gleichung kann die Ausgabegröße nach der Faltung für ein quadratisches Eingabebild berechnet werden:

$$\text{Outputsize} = \frac{\text{Inputsize} - \text{Kernel size} + 2 * \text{Padding}}{\text{Stride}} + 1$$

Aufgrund der Einschränkung auf quadratischen Bildern beschreibt die Ein- und Ausgabegröße immer sowohl Höhe als auch Breite des Bildes.

## 2. Aktivierungsschicht:

Eine Aktivierungsschicht meint eine Neuronen-Schicht, die eine Aktivierungsfunktion

auf die eingehenden Daten anwendet. Das Format der Daten ändert sich dabei nicht. In der Literatur ist nicht immer die Rede von einer Aktivierungsschicht. Häufig gehen die Autoren davon aus, dass eine Aktivierungsfunktion nach dem Filterungsprozess innerhalb der Neuronen jeder Faltungsschicht angewendet wird. Die ReLU-Aktivierungsfunktion ist beim Merkmals-Lernen weit verbreitet [5, 17, 34].

### 3. Batch-Normalisierungs-Schicht:

Es ist vorteilhaft, die von einer Faltungs- oder Aktivierungsschicht verarbeiteten Daten zu normalisieren. Dadurch kann verschwindenden und explodierende Gradienten vorbeugt werden, die das Training von Neuronalen Netzen fehlschlagen lassen. Mit der „Batch-Normalisierung“ werden Daten in ein für das Training des CNN optimales Intervall überführt [5]. Das Anwenden der Batch-Normalisierung wird genau wie das Anwenden einer Aktivierungsfunktion nicht immer als separate Schicht, sondern als Teil einer Faltungsschicht angesehen.

Für eine Batch-Normalisierung werden die Trainingsdaten auf mehrere Gruppen (Batches) aufgeteilt. Über jeden Batch wird separat das Arithmetische Mittel  $\mu$  und die Varianz  $\sigma$  gebildet. Anschließend werden die Datensätze  $v_i$  des Stapels wie folgt normalisiert [5]

$$\hat{v}_i = \frac{v_i - \mu}{\sigma}.$$

Nach der Normalisierung werden die Daten bei der Batch-Normalisierung noch verschoben. Dazu werden zwei variable Parameter  $\gamma$  und  $\beta$  eingesetzt. Im Trainingsprozess werden  $\gamma$  und  $\beta$  ähnlich wie die Gewichte laufend angepasst, um Unterschiede in den Datensätzen auszugleichen. Beide Parameter unterscheiden sich für jeden Batch. Mit dem folgenden Term wird aus dem normalisierten  $\hat{v}_i$  Datensatz das verschobene Ergebnis  $a_i$  der Batch-Normalisierung [5]:

$$a_i = \gamma \cdot \hat{v}_i + \beta.$$

### 4. Bündelungsschicht:

Bündelungsschichten dienen der Verdichtung und Reduktion von Datenmengen. Eine Bündelungsschicht folgt in einem CNN i. d. R. auf eine Faltungsschicht. Bündelungsschichten arbeiten auf den erstellten Merkmalskarten. Eine Bündelungsschicht nutzt ein quadratisches Bündelungs-Feld, um mehrere Datenpunkte zu einem zu verdichten. Dieses Bündelungs-Feld wird genau wie die Filtermuster einer Faltungsschicht mit einer wählba-

ren Schrittweite (Stride) über eine Merkmalskarte geschoben. Dabei treten Bündelungsfelder nicht über die Ränder der Merkmalskarten hinaus. Daher wird hier auch kein Padding benötigt.

Im Gegensatz zu Filtermustern enthalten Bündelungsfelder keine Gewichte und geben nur einen Bereich vor, in dem die Datenpunkte der Merkmalskarte verdichtet werden sollen. Das Verdichten kann mit verschiedenen „Bündelungsfunktionen“ erreicht werden. Die Bündelungsfunktion gilt dabei für die gesamte Bündelungsschicht. Bündelungsfunktionen sind i. d. R. nichtlineare Aggregatfunktionen. Beispiele für Bündelungsfunktionen sind das „Max-Pooling“, bei dem das Ergebnis der größte Wert eines Bündelungsfelds ist und das „Avg-Pooling“, bei dem das Ergebnis der Durchschnitt aller Werte eines Bündelungsfelds ist. Das Max-Pooling führt i. d. R. bei Bilddaten zu besseren Ergebnissen als das Avg-Pooling [5, 24, 28].

Die Größe des Bündelungsfelds und die Schrittweite werden bei der Konstruktion eines CNN für jede Bündelungsschicht separat ausgewählt. Die Faktoren können so gewählt werden, dass Teile der Merkmalskarte nicht abgedeckt und somit von dem CNN nicht weiter betrachtet werden.<sup>2</sup> Es ist ebenso möglich, dass die Teilbereiche sich überschneiden. Bei der Wahl der Faktoren „Größe des Bündelungsfelds“ und „Schrittweite“ muss das Format und die Größe der zu bündelnden Merkmalskarte mit einbezogen werden. Meistens werden Bilder bereits vor dem Einspeisen in ein CNN in ein passendes Format (oft quadratisch) zugeschnitten [5, 6].

In der Praxis hat sich gezeigt, dass Merkmalskarten mit Schrittweiten von größer als 1 verkleinert werden können, ohne dabei klassifizierungsrelevante Informationen zu entfernen. Die Seitenlänge  $x$  der Teilbereiche von einer zu bündelnden Merkmalskarte mit Seitenlänge  $l$  befindet sich i. d. R. in dem Bereich  $1 < x < l$  [5, 24]. Die Kombination aus einem Bündelungsfeld der Größe  $2 \times 2$  und der Schrittweite 2 ist weit verbreitet. Abbildung 2.7 zeigt eine Merkmalskarte der Größe  $6 \times 6$ , die mit einem Bündelungsfeld der Größe  $2 \times 2$ , einer Schrittweite von 2 und dem Max-Pooling zu einem Datenfeld der Größe  $3 \times 3$  verdichtet wird.

---

<sup>2</sup>Das ist z. B. der Fall, wenn eine Merkmalskarte der Größe  $5 \times 5$  mit einem Bündelungsfeld der Größe  $2 \times 2$  und der Schrittweite 2 verdichtet werden soll. Denn Bündelungsfelder dürfen nicht über den Rand einer Merkmalskarte hinaus treten.

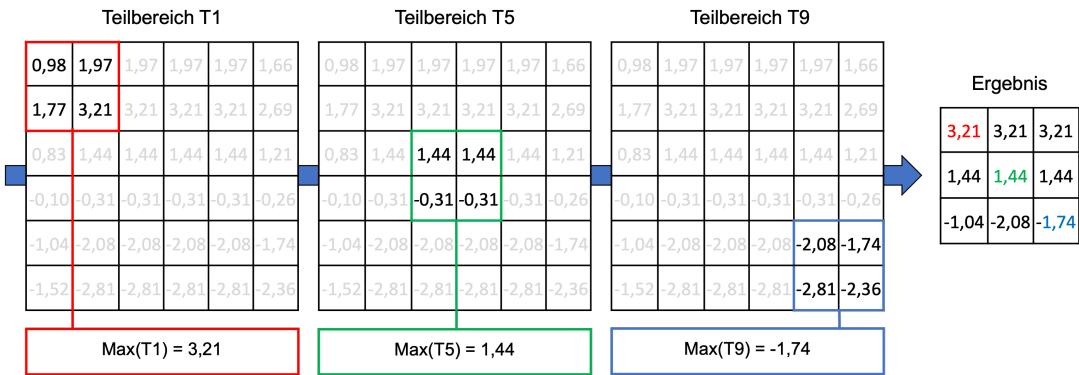


Abbildung 2.7.: Der Verdichtungsprozess einer Bündelungsschicht:

Eine Merkmalskarte der Größe  $6 \times 6$  wird mit einem Bündelungs-Feld der Größe  $2 \times 2$ , einer Schrittweite von 2 und dem Max-Pooling zu einem Datenfeld der Größe  $3 \times 3$  verdichtet.

CNNs haben in diesem Anwendungsfall alleine durch die spezielle Eingabeschicht deutlich weniger Neuronen als klassische KNN. Mit den Schichten für das Merkmals-Lernen wird die Anzahl der zu trainierenden Gewichte im Vergleich zu einem klassischen KNNs reduziert, da die Schichten eines CNN nicht vollverknüpft sind. Zudem werden die Filtermuster und damit die Anzahl der Gewichte i. d. R. kleiner, je weiter hinten die Faltungsschicht liegt. Denn durch die Bündelungsschichten werden die Merkmalskarten immer weiter verkleinert. Dadurch können CNNs deutlich effizienter in Bezug auf den Speicherbedarf und Rechenaufwand trainiert werden, als klassische KNNs. Zusätzlich erzielen CNNs höhere Vorhersage-Genauigkeit als klassische KNNs [5, 6, 34].

Die Schichten für das Merkmals-Lernen bestehen zu einem Großteil aus Paaren von einer Faltungsschicht inklusive ReLu-Aktivierungsfunktion und einer Bündelungsschicht. Dabei unterscheiden sich meist die Schichten durch die relevanten Parameter („Anzahl Filtermuster“, „Schrittweite“, „Größen und Formate von Datenfeldern“) voneinander. CNNs haben neben den Paaren aus Faltungs- und Bündelungsschichten häufig auch längere Folgen von Faltungsschichten. Bündelungsschichten könnten theoretisch auch direkt aufeinander folgen. In der Praxis tritt dies jedoch nur selten auf, da dadurch die Daten zu schnell verdichtet werden und klassifizierungsrelevante Informationen verloren gehen können.

CNNs können genau wie klassische KNNs noch viele weitere Schicht-Arten beinhalten, die an dieser Stelle nicht weiter behandelt werden. Die Schichten werden i. d. R. so

konfiguriert, dass die Anzahl Merkmalskarten zum Ende hin größer und die Größe der Merkmalskarten kleiner wird. Auf diese Weise werden die umfangreichen mehrdimensionalen Daten zu einer eindimensionalen Liste von verdichteten Informationen, die mit klassischen KNN-Schichten verarbeitet werden können [5, 17, 24].

Die versteckten Schichten für das Merkmals-Lernen liefern die Informationen, mit denen die folgenden versteckten, vollverknüpften Schichten eine Klassifizierung durchführen. Die versteckten Schichten für das Klassifizieren unterscheiden sich nicht von den versteckten Schichten eines klassischen Klassifizierungs-KNN (siehe Kapitel 2.1) [5, 17].

Die Ausgabeschicht eines CNN unterscheidet sich nicht von der Ausgabeschicht eines klassischen KNN-Klassifizierers. Die Ausgabeschicht besitzt so viele Neuronen wie es Klassen gibt. Das Ausgabe-Neuron mit dem höchsten Aktivierungszustand gibt an, welcher Klasse ein Bild von dem CNN zugeordnet wurde. Dazu wird häufig die Softmax-Aktivierungsfunktion verwendet. Für binäre Klassifizierungsaufgaben wird stattdessen meist die Sigmoid-Funktion eingesetzt [5, 17].

## 2.3. Generative Adversarial Networks

Ein Generative Adversarial Network (GAN) ist ein Machine-Learning Modell zur Generierung von Daten. Ein bekannter Anwendungsfall von GANs ist das Erzeugen möglichst realistisch aussehender Bilder oder Videos. Dafür müssen GANs strukturell völlig anders aufgebaut sein. Zudem werden neuartige Optimierungs- oder Loss-Funktion für GANs verwendet.

Die Idee und der Zweck von GANs werden an einem einfachen Beispiel deutlich. Soll ein Klassifizierer dafür trainiert werden Katzenbilder von Nicht-Katzenbildern zu unterscheiden, so werden gelabelte Trainingsdaten benötigt. Das Label gibt für jeden Trainingsdatensatz an, ob auf dem Bild eine Katze abgebildet ist oder nicht. Solche Datensätze zu erstellen, ist sehr aufwändig. Denn es müssen viele Bilder für beide Klassen manuell gesammelt und gelabelt werden. Mit Hilfe von GANs können die notwendigen Bilder synthetisch erzeugt und automatisch gelabelt werden.

GANs bestehen aus zwei KNNs (siehe Abbildung 2.8): einem Diskriminatoren und einem Generator. Der Diskriminatoren ist ein Klassifizierungs-KNN, mit der Aufgabe zu ent-

scheiden, ob ein Bild echt oder synthetisch ist. Der Generator ist ein KNN, das Bilder erzeugt, die den Diskriminatoren bei seiner Entscheidung täuschen sollen. Wenn der Generator durch das Training besser wird und realistischere Bilder generieren kann, wird es für den Diskriminatoren gleichzeitig schwieriger, diese synthetischen Bilder von den echten zu unterscheiden. Durch dieses Gegenspiel wird der Diskriminatoren mit der Zeit immer besser darin, echte Bilder von synthetischen zu unterscheiden und der Generator lernt, täuschend echte Bilder zu erzeugen. Das Training ist abgeschlossen, wenn die erzeugten Bilder nicht mehr von echten Bildern zu unterscheiden sind.

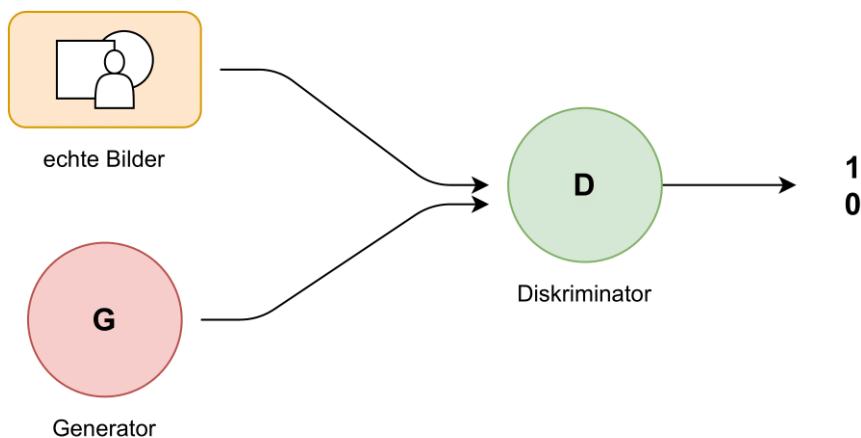


Abbildung 2.8.: Die Funktionsweise eines GAN:

Dem Diskriminatoren werden sowohl echte Bilder als auch generierte Bilder gezeigt. Der Diskriminatoren muss entscheiden, ob ein Bild echt oder synthetisch ist. Der Generator erzeugt die besagten synthetischen Bilder und versucht den Diskriminatoren zu täuschen.

In der Spieltheorie wird dieses Gegenspiel als Minimax-Spiel bezeichnet. In seinem Paper zu GANs beschreibt Ian Goodfellow das Minimax-Spiel mit der Formel [36]

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

Der Generator möchte das Ergebnis der Funktion  $V$  minimieren, während der Diskriminatoren das Ergebnis maximieren will. Der Wert der Funktion  $V$  setzt sich aus den Erwartungswerten für alle  $x$  aus der Wahrscheinlichkeitsverteilung der Trainingsdaten und alle  $z$  aus einer beliebig wählbaren Verteilung für die Startwerte zusammen. Der Loss-Wert für jedes  $x$  bzw.  $z$  aus der entsprechenden Verteilung wird von Goodfellow mithilfe der binären Kreuzentropie berechnet. In der Theorie konvergiert das GAN genau dann, wenn

Diskriminator und Generator ein Nash-Gleichgewicht erreichen. Das Nash-Gleichgewicht ist eine Kombination von Strategien für nicht-kooperative Spieler, bei denen es für keinen Spieler Sinn ergibt, als einziger eine andere Strategie zu wählen. Wenn das Nash-Gleichgewicht erreicht ist, wurde damit der optimale Punkt für die Minimax-Gleichung gefunden [37]. In einem solchen Fall entscheidet der Diskriminat or nur durch Zufall, ob ein Bild echt oder synthetisch ist.

Um das beschriebene Vorgehen beim Training eines GAN umzusetzen, muss betrachtet werden, welche Aktionen beim Diskriminat or bestraft bzw. belohnt werden sollen:

- Der Diskriminat or wird belohnt, wenn er ein echtes Bild als echt klassifiziert.
- Der Diskriminat or wird bestraft, wenn er ein echtes Bild als synthetisch klassifiziert.
- Der Diskriminat or wird bestraft, wenn er ein synthetisches Bild als echt klassifiziert.

Das Ziel des Generators ist es, Bilder zu generieren, die von den echten Bildern nicht unterscheidbar sind. Aus diesem Ziel folgt, dass folgende Aktionen bestraft bzw. belohnt werden müssen:

- Der Generator wird belohnt, wenn der Diskriminat or ein synthetisches Bild nicht als solches erkennt.
- Der Generator wird bestraft, wenn der Diskriminat or ein synthetisches Bild als solches erkennt.

Damit der Trainingsprozess eines GAN funktioniert, müssen der Diskriminat or und Generator gleichzeitig lernen. Wenn zuerst der Diskriminat or und anschließend der Generator mit allen Trainingsdaten trainiert wird, würde der Generator nicht lernen können. Denn dann erledigt der Diskriminat or seine Aufgabe wesentlich besser als der Generator. Das Training von GANs funktioniert nur dann, wenn Diskriminat or und Generator sich stetig in einem Gleichgewicht befinden und keiner der beiden zu stark ist. Deshalb müssen in jeder Iteration der Trainingsschleife sowohl der Diskriminat or als auch der Generator trainiert werden. Beim Diskriminat or und Generator gibt es in Summe drei Aktionen, die während des Trainings bestraft werden müssen. Daher besteht die Trainingsschleife aus drei Schritten.

1. Im ersten Schritt wird dem Diskriminator ein echtes Bild gezeigt. Der Diskriminator bekommt zudem die Information, dass es sich um ein echtes Bild handelt. Das Ergebnis der Klassifizierung soll dementsprechend 1,0 betragen. Die Abweichung der Vorhersage des Diskriminators von diesem erwarteten Ergebnis wird verwendet, um den Diskriminator zu verbessern.
2. Anschließend wird mit dem Generator ein Bild generiert. Dieses synthetische Bild wird dem Diskriminator zusammen mit der Information übergeben, dass es sich um ein synthetisches Bild handelt. Das Ergebnis der Klassifizierung soll dementsprechend 0,0 betragen. Die Abweichung der Vorhersage des Diskriminators von diesem erwarteten Ergebnis wird verwendet, um den Diskriminator ein weiteres Mal zu aktualisieren. Der Generator darf in diesem Schritt nicht aktualisiert werden. Denn der Generator soll nicht dafür belohnt werden, dass ein synthetisches Bild als solches erkannt wird.
3. Im letzten Schritt wird der Generator trainiert. Dafür wird wieder ein Bild mit dem Generator erzeugt und dem Diskriminator gezeigt. Das Ziel des Generators ist es, ein Bild zu erzeugen, das der Diskrimantor als echt klassifiziert. Das erwartete Ergebnis der Klassifizierung beträgt dementsprechend 1,0. Die Abweichung der Vorhersage des Diskriminators von diesem erwarteten Ergebnis wird verwendet, um den Generator zu verbessern.

### Schwierigkeiten beim Training von GANs

In der Praxis ist das Training von GANs eine schwierige Aufgabe. Der Diskriminator und der Generator müssen sich stets in einem Gleichgewicht befinden, damit beide lernen. Dieses Gleichgewicht herzustellen, ist kompliziert. Das Konzept von GANs befindet sich Anfang 2022 noch in einem Anfangsstadium. Daher ist nicht genau bekannt, wann das Training von GANs scheitert oder erfolgreich ist. Die drei größten Probleme beim Training eines GAN sind:

1. Das Mode Collapse Problem: Der Generator lernt, unabhängig von der Eingabe immer das gleiche Bild zu generieren.

2. Divergenz der Parameter: Der Gradientenabstieg ist nicht immer ein gutes Verfahren, um die besten Parameterbelegungen in einem Minimax-Spiel zu finden. Anstatt konvergierender Parameter kann es durch den Gradientenabstieg bei einem Minimax-Spiel zu divergierender Parameter kommen.
3. Verschwindende Gradienten: Wenn der Diskriminatior zu stark wird, kommt es zu verschwindenden Gradienten und der Generator kann nicht mehr lernen.

### Mode Collapse

Beim Mode Collapse Problem lernt der Generator nicht mehr, verschiedene Bilder zu generieren und erzeugt unabhängig von der Eingabe immer das gleiche Bild. Am Beispiel des MNIST-Datensatz würde das bedeuten, dass der Generator lernt, z. B. nur noch eine 3 zu erzeugen. Die Gründe für den Mode Collapse sind in der Forschung noch nicht vollständig bekannt und werden aktuell untersucht. Eine mögliche Erklärung für das Problem ist, dass der Generator in einem frühen Stadium des Trainings ein Bild gefunden hat, mit dem der Diskriminatior immer getäuscht werden konnte [6]. In der Forschung wird davon ausgegangen, dass GANs die Wahrscheinlichkeit für das Auftreten von Merkmalen aus den Trainingsdaten erlernen [6]. Diese Hypothese vertritt auch Ian Goodfellow [36].

Die Schwierigkeit liegt darin, dass der Generator die Wahrscheinlichkeitsverteilungen aller in den Trainingsdaten enthaltenen Klassen erlernen muss. Allerdings existiert zur Zeit keine GAN-Architektur, mit der das perfekt umgesetzt wird [38]. Ein teilweiser Mode Collapse ist in der Praxis deshalb nicht zu vermeiden, auch wenn dieser nicht auffällt. Nur ein totaler Mode Collapse, bei dem aus jedem Startwert immer das gleiche Bild resultiert, kann festgestellt werden.

Ein totaler Mode Collapse kann bei der Entwicklung eines GAN nicht im Vorhinein vermieden werden. Zudem gibt es keine eindeutige Lösung zur Behebung des Mode Collapse, wenn dieser erst im Laufe des Trainings auftritt. In der Praxis haben sich zwei Methoden bewährt, mit denen dem Mode Collapse entgegen gewirkt werden kann. Die erste Möglichkeit ist das Anpassen Struktur von Generator und Diskriminatior. GANs reagieren meist sehr empfindlich auf diese Änderungen, weshalb durch das Hinzufügen, Löschen oder Verändern von Schichten dem Mode Collapse vergebeugt werden kann.

Insbesondere eine Normalisierung der Signale zwischen verschiedenen Schichten kann eine Verbesserung bringen [6] und wird deshalb in der Praxis häufig eingesetzt. Durch die Normalisierung bewegen sich die Signale mit einer begrenzten Varianz um den Mittelwert Null. Die zweite Möglichkeit, um einem Mode Collapse vorzubeugen, sind normalisierte Startwerte. In der Forschung wurde bereits gezeigt, dass eine Normalisierung der Startwerte generell zu einer Verbesserung des Trainings führt [6, 39].

### Divergenz der Parameter

Das zweite große Problem beim Training von GANs ist die Divergenz der Parameter. Der Gradientenabstieg ist ein gut erforschtes Verfahren zur Minimierung einer Funktion. Es existieren ausgereifte Varianten des Gradientenabstiegs-Verfahrens wie zum Beispiel das ADAM-Verfahren. Jedoch scheint der Gradientenabstieg kein guter Ansatz zur Lösung des Minimax-Spiels zu sein. Anhand der einfachen Funktion  $f = x * y$  kann dieses Problem besser veranschaulicht werden.

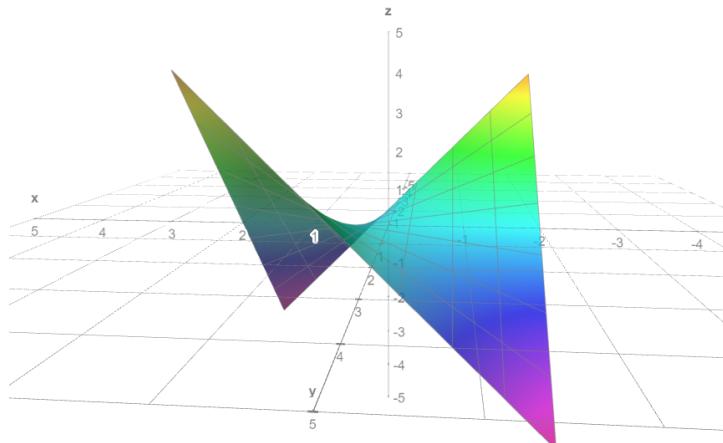


Abbildung 2.9.: Minimax-Spiele mit Gradientenabstiegs-Verfahren bearbeiten:

Die Oberfläche der Funktion  $f = x * y$  sieht aus wie ein Sattel. Intuitiv liegt die beste Lösung für das Minimax-Spiel der Funktion bei  $(x, y) = (0, 0)$ . Mit einem Gradientenabstiegs-Verfahren wird dieser Punkt nicht gefunden.

Die Oberfläche der Funktion  $f(x, y)$  stellt einen Sattel dar. Wenn es zwei Spieler gibt und einer die Funktion  $f = x * y$  minimieren möchte, während der andere die Funktion maximieren möchte, so liegt die beste Lösung bei  $(x, y) = (0, 0)$ . Jeder Spieler kann nur

einen Parameter beeinflussen, um sein Ziel zu erreichen. Nur für  $(x, y) = (0, 0)$  können sich beide Spieler sicher sein, dass der Gegner das Ergebnis nicht mehr beeinflussen kann. Setzt der erste Spieler z. B.  $x = 2$  und der zweite Spieler  $y = 0$ , so ist das so ergibt sich  $f(x, y) = 0$ . Das ist das gleiche Ergebnis wie für  $(x, y) = (0, 0)$ . Möchte der zweite Spieler das Ergebnis aber nicht minimieren sondern maximieren, könnte dieser für  $y$  einen Wert größer Null wählen. Dadurch würde der zweite Spieler seinem Ziel der Ergebnis-Maximierung näher kommen und der erste Spieler würde sich von seinem Ziel der Ergebnis-Minimierung entfernen. Zudem liefert die Lösung  $(x, y) = (0, 0)$  genau die Mitte der möglichen Funktionswerte für  $f$ . Deshalb stellt dieses Ergebnis beide Spieler in gleichem Maße zufrieden [6].

Beim Gradientenabstieg wird versucht das optimale Ergebnis zu finden, indem die Parameter des Modells durch den Anstieg des Gradienten angepasst werden. Dafür werden die folgenden beiden Gleichungen mit der Lernrate  $\alpha$  verwendet:

$$\begin{aligned} x &= x + \alpha * \frac{\partial f}{\partial x} \\ y &= y - \alpha * \frac{\partial f}{\partial y}. \end{aligned}$$

In diesem Minimax-Spiel soll  $f(x, y)$  durch den Parameter  $x$  maximiert werden. Dazu muss der Gradient zu  $x$  addiert werden. Durch den Parameter  $y$  soll die  $f(x, y)$  minimiert werden. Dazu muss der Gradient von  $y$  subtrahiert werden. Durch Berechnen der partiellen Ableitungen ergeben sich folgenden Aktualisierungsregeln für jede Iteration:

$$x = x + \alpha * y$$

$$y = y - \alpha * x.$$

Mit diesen Berechnungsvorschriften können die Werte der Parameter  $x$  und  $y$  in jedem Iterationsschritt des Gradientenabstiegs-Verfahrens errechnet werden. Die Entwicklung der Parameter ist in Abbildung 2.10 zu sehen. Idealerweise würden  $x$  und  $y$  beide gegen Null konvergieren, um so das Nash-Gleichgewicht zu finden. In Abbildung 2.10 ist jedoch eine Divergenz mit einer immer größer werdenden Amplitude zu erkennen. Das bedeutet im Laufe des Trainings entfernen sich die Parameter immer weiter von der optimalen Lösung. Dieses Beispiel zeigt sehr eindrucksvoll, dass ein Gradientenabstiegs-Verfahren schlecht für das Training von GANs geeignet ist [6, 37].

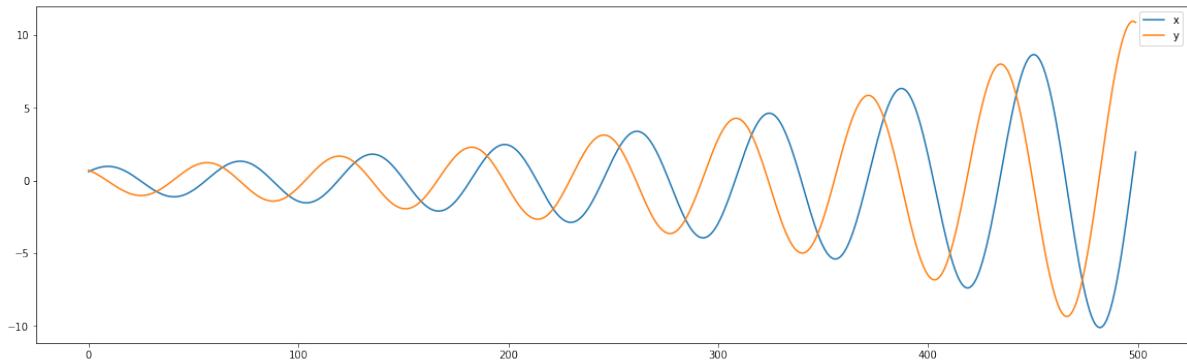


Abbildung 2.10.: Divergenz der Parameter bei Gradientenabstiegs-Verfahren im Kontext von Minimax-Spielen:

Um einen Optimalen Punkt  $(x, y)$  in dem Minimax-Spiel mit der Funktion  $f(x, y) = x * y$  zu finden, kann kein einfaches Gradientenabstiegs-Verfahren verwendet werden. Denn die Parameter  $x$  und  $y$  divergieren mit fortschreitendem Training.

In der Praxis werden dennoch mit Gradientenabstiegs-Verfahren gute Lösungen für solche Probleme gefunden. Das liegt an der Komplexität der GAN-Modelle. Durch eine große Anzahl an Parametern ergibt sich eine komplexe Loss-Oberfläche mit vielen lokalen Minima und Maxima. Dadurch ist die Wahrscheinlichkeit für eine Divergenz geringer [6]. In der Praxis gibt es Anfang 2022 kaum eine Alternative zu Gradientenabstiegs-Verfahren, weshalb diese Verfahren bei GANs noch immer der Standard sind. Die Divergenz der Parameter kann durch die Auswahl der Loss-Funktion beeinflusst werden. Der Schwerpunkt der Forschung liegt daher auf dem Finden neuer Loss-Funktionen [6, 37].

### Verschwindende Gradienten

Ein weiteres Problem beim Training von GANs sind verschwindende Gradienten. Beim Generator wird in der Output-Schicht immer eine sigmoide Aktivierungsfunktion genutzt. Diese Einschränkung resultiert aus der Art und Weise wie Bilder dargestellt werden. Zur Darstellung von Farbbildern wird häufig der RGB-Farbraum mit einem Wertebereich von 0 bis 1 oder -1 bis 1 verwendet. Die Sigmoid-Funktion und der Tangens Hyperbolicus sind zwie Aktivierungsfunktionen, die reelle Werte in diesen Wertebereich abbilden. Bei Klassifizierungsproblemen werden die Ergebnisse üblicherweise als diskrete

Wahrscheinlichkeitsverteilung angegeben. Daher muss der Diskriminator Werte im Bereich von 0 bis 1 produzieren. Deshalb wird für den Diskriminator in der Output-Schicht die Sigmoid-Aktivierungsfunktion genutzt.

Ein Problem mit sigmoiden Aktivierungsfunktion sind verschwindende Gradienten, wenn der Eingabewert sehr klein oder sehr groß ist. Dieses Verhalten stellt ein Problem beim Training von GANs dar, wenn der Diskriminator zu stark wird. Ist der Diskriminator sehr gut im Erfüllen seiner Aufgabe, liefert dieser eine eindeutige und starke Antwort. Diese Sicherheit bedeutet, dass der Eingabewert in die Aktivierungsfunktion sehr groß und der Gradient damit sehr klein ist.

Um dieses Problem zu mildern, wird bei GANs i. d. R. nur auf den Ausgabeschichten von Generator und Diskriminator eine sigmoide Aktivierungsfunktion verwendet. Denn die genannten Einschränkungen der Wertebereiche gelten nur für die Ausgabeschichten. Würde jede Schicht eine sigmoide Aktivierungsfunktion verwenden, so würde der Gradient mit jeder Schicht kleiner werden und gegen Null konvergieren. Daher wird in den restlichen Schichten die Leaky ReLU Aktivierungsfunktion eingesetzt. Die Leaky ReLU Funktion bietet den Vorteil, dass an jeder Stelle ein Gradient mit einer Steigung ungleich Null vorhanden ist. Kommt es zu dem Fall, dass der Diskriminator sehr stark ist und der Gradient nur noch eine kleine Steigung besitzt, so wird zumindest diese kleine Steigung im Netz zurück geleitet. Zusätzlich zur Leaky ReLU Funktion wird in der Praxis die Batch-Normalisierung verwendet, um verschwindende Gradient zu vermeiden. Die Batch-Normalisierung normalisiert die Eingabewerte zwischen den versteckten Schichten eines KNNs.

Das Problem der verschwindenden Gradienten kann mit einer geeigneten Loss-Funktion behoben werden. Mit der Wasserstein-Distanz kann zusätzlich einer Divergenz der Parameter entgegen gewirkt werden. Wasserstein-GANs (WGANs) sind daher stabiler trainierbar. WGANs konnten sich in der Praxis beweisen und werden in vielen Anwendungsfällen erfolgreich eingesetzt.

## **Wasserstein-GAN**

Das Wasserstein-GAN (WGAN) Konzept wurde im Jahr 2017 von Martin Arjovsky vorgestellt und ist resistenter gegen alle drei aufgeführten Probleme [40]. Ebenfalls gibt

ein WGAN mit dem Loss-Wert Informationen über die Konvergenz von Generator und Diskriminator. Bei der klassischen GAN-Architekturen ist das anhand des Loss-Werts nicht erkennbar [40].

In der Theorie sollen generative Modelle beim Training mit Daten aus einer unbekannten Verteilung  $P_{real}$  lernen. Das Modell soll die Verteilung  $P_{real}$  durch die Verteilung  $P_\theta$  approximieren, wobei  $\theta$  die Parameter der Verteilung sind. Bei der ursprünglichen GAN-Architektur wurde die Verteilung  $P_\theta$  durch ein Maximum-Likelihood Verfahren optimiert. WGANs verfolgen einen anderen Ansatz. Anstatt die Verteilung zu lernen, wird bei WGANs eine differenzierbare Funktion  $g_\theta$  erlernt, die die Verteilung  $Z$  in die Verteilung  $P_\theta$  transformiert [40].  $Z$  ist dabei die Verteilung der Startwerte, wobei die Werte einer beliebigen Verteilung entstammen können.

Um die Funktion  $g_\theta$  unter Berücksichtigung der Parameter  $\theta$  zu lernen, wird eine Distanz zwischen den beiden Verteilungen benötigt. Es gibt verschiedene Metriken zur Bestimmung der Distanz zweier Wahrscheinlichkeitsverteilungen. Beispiele sind die Kullback-Leibler Divergenz oder die Jenson-Shannon Divergenz [40]. Das Paper zeigt jedoch, dass es für die Verteilung  $Z$  Fälle geben kann, in denen nur die Wasserstein Distanz konvergiert [40].

Bei der Wasserstein-Distanz wird die Wahrscheinlichkeitsverteilung darüber definiert, wie viel „Masse“ jedem Punkt der Verteilung zugeordnet ist. Wenn  $\Pi(P_r, P_g)$  das Set aller multivariaten Verteilungen  $\gamma$  mit den Randverteilungen  $P_r$  und  $P_g$  ist, dann wird die Wasserstein-Distanz durch folgende Gleichung definiert:

$$W(P_r, P_g) = \inf_{\gamma \in \Pi(P_r, P_g)} \mathbb{E}_{x,y \sim \gamma} [\|x - y\|]. \quad (2.2)$$

Die Wasserstein-Distanz bemisst die minimale Anstrengung die nötig ist, um die Masse einer Verteilung in die andere zu transportieren. Jedes  $\gamma$  kann als eine Art Transportplan interpretiert werden.  $\gamma(x, y)$  gibt für jede Masse die Kosten für den Transport von  $x$  nach  $y$  an. In der Implementierung wird eine Approximation verwendet, bei der das Supremum über alle 1-Lipschitz stetigen Funktionen ausgewählt wird:

$$W(P_r, P_\theta) = \sup_{\|D\|_L \leq 1} \mathbb{E}_{x \sim P_r} [D(x)] - \mathbb{E}_{x \sim P_\theta} [D(x)]. \quad (2.3)$$

Aus dieser Approximation ergibt sich eine Einschränkung für den Diskriminator  $D(x)$ . Um die Lipschitz-Beschränkung zu implementieren, müssen die Gewichte des Diskriminators nach jeder Iteration im Training auf ein Intervall  $[-c, c]$  beschränkt werden. Dieses Beschränken der Gewichte im Training wird als „Weight Clipping“ bezeichnet [40]. In der Praxis hat sich das Intervall  $[-0.01, 0.01]$  als eine gute Wahl herausgestellt [40]. Aufgrund der Wasserstein-Distanz wird der Diskriminator umbenannt und heißt bei WGANs „Critic“ [41]. Die Wasserstein-Distanz gibt einen Wert an, der als Maß für die Echtheit eines Bildes interpretiert werden kann. Der Diskriminator äußert somit eine Kritik. Es findet keine Diskriminierung zwischen echten und synthetischen Bildern mehr statt.

Durch die Einschränkung des Weight Clippings wird die Fähigkeit, komplexe Funktionen zu modellieren, eingeschränkt. Aus diesem Grund entstanden WGANs mit Gradient Penalty [41]. Anstatt die Gewichte zu beschränken, wird bei diesem Ansatz die Norm des Gradienten bestraft. Ziel ist, dass die Norm des Gradienten 1 beträgt [41]. Dadurch lernt der Critic selbst die Bestrafung möglichst gering zu halten. In Tests wurde gezeigt, dass mit diesem Vorgehen Bilder mit besserer Qualität generiert werden können [41].

### Conditional Generative Adversarial Networks (CGANs)

Bisher wurde darauf eingegangen, wie GANs aufgebaut sind, welche Probleme es beim Training gibt und welche Verbesserungen bereits entwickelt wurden. Allerdings gibt es bei den bisher vorgestellten Architekturen keine Möglichkeit, die Ausgabe des Generators zu beeinflussen. Die generierten Bilder sind immer zufällig. Mithilfe von sogenannten Conditional Generative Adversarial Networks (CGANs) kann jedoch beeinflusst werden, wie das generierte Bild aussieht.

Um den Generator so zu trainieren, dass dieser ein Bild aus einer bestimmten Klasse erzeugt, wird der zufällige Startwert um ein Label erweitert (siehe Abbildung 2.11). Bei dem Diskriminator muss die Eingabe ebenfalls um das Label erweitert werden (siehe Abbildung 2.11). Es kann nicht wie bei einer klassischen Klassifikation das Bild als einzige Eingabe dienen und das Label als erwartetes Ergebnis. Denn der Diskriminator soll nicht unterscheiden, ob ein gegebenes Bild echt oder synthetisch ist. Es soll nicht nach dem Label klassifiziert werden. In der Trainingsschleife muss berücksichtigt werden, dass das zufällige Label dem Generator und dem Diskriminator übergeben werden muss.

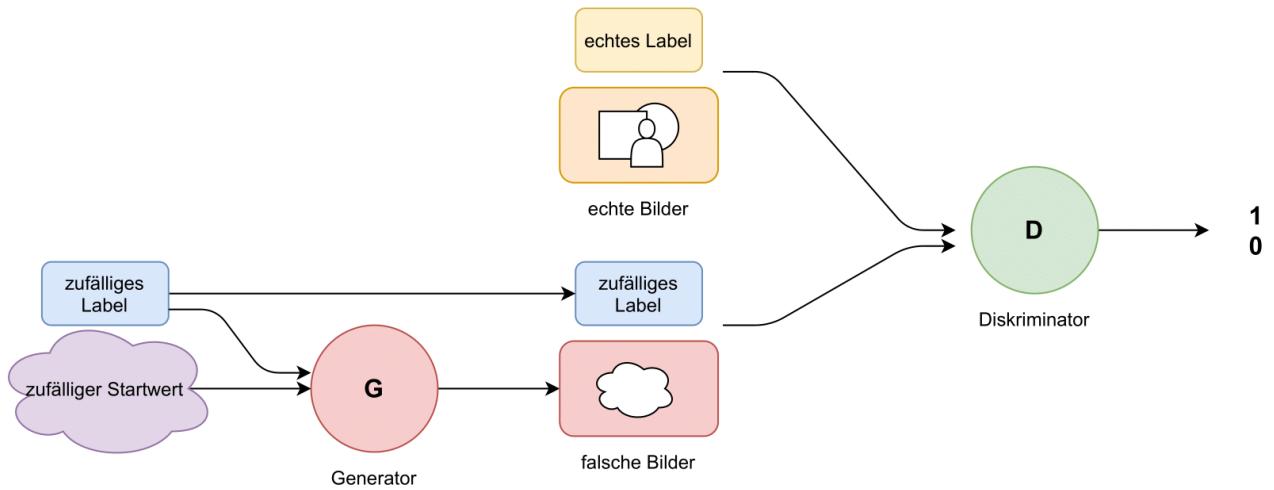


Abbildung 2.11.: Die Architektur eines CGAN:

Neben einem zufälligen Startwert wird dem Generator ein Label mitgegeben, um ein Bild mit spezifischem Inhalt zu erzeugen. Dem Diskriminatator muss das Label zusammen mit einem Bild als Eingabe und nicht als erwartetes Ergebnis übergeben werden.

Durch diese Anpassungen können gezielt Bilder aus unterschiedlichen Klassen generiert werden.

### Grundlagen für die Implementierung generativer Modelle

Im vorherigen Kapitel wurden bereits Faltungsschichten für die Bildklassifizierung vorgestellt. Faltungsschichten bieten eine effiziente Möglichkeit, Merkmale aus den Bildern zu extrahieren. Für generative Modelle, wie einem GAN, ist eine umgekehrte Vorgehensweise nötig. Die Eingabewerte sind die Merkmale und aus diesen Merkmalen soll das Bild generiert werden. Die einfachste Möglichkeit, um aus z. B. 100 Eingabewerten die benötigte Anzahl an Pixelwerten zu erzeugen, sind vollverknüpfte Schichten. Dieser Ansatz ist für große Datenmengen jedoch nicht effizient. Eine effiziente Möglichkeit, um aus gegebenen Merkmalen ein Bild zu erzeugen, ist die transponierte Faltung (siehe Abbildung 2.12). Die transponierte Faltung ist die Umkehrung der Faltung und ermöglicht das Vergrößern eines Bildes.

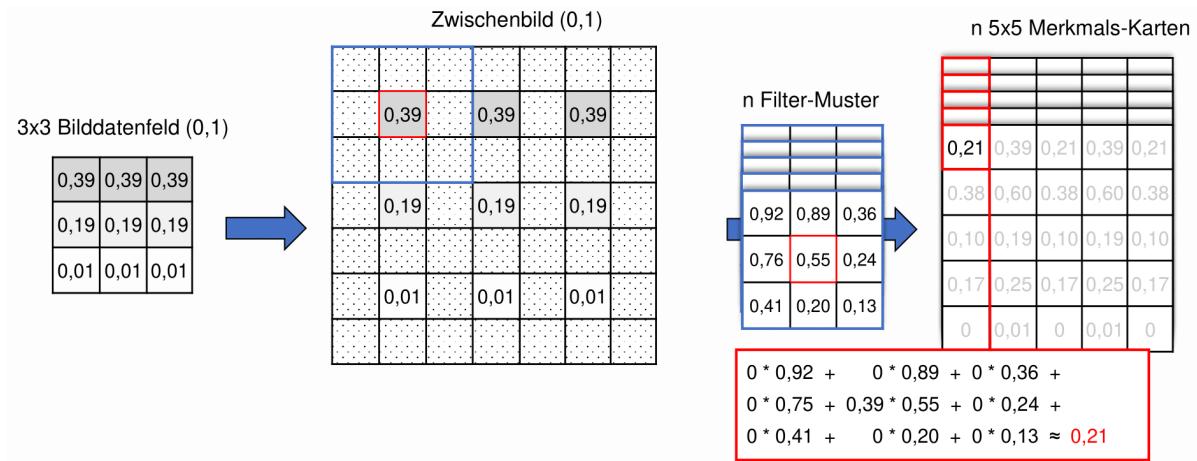


Abbildung 2.12.: Funktionsweise der transponierten Faltung:

Als erstes wird eine Zwischenbild erstellt, indem leere Pixel in der Eingabe eingefügt werden. Anschließend wird eine Faltung ausgeführt, so dass die leeren Pixel Farbwerte auf Basis der umliegenden Pixel erhalten.

Bei der transponierten Faltung wird im ersten Schritt ein Zwischenbild erstellt. Bei diesem Zwischenbild werden Pixel mit dem Wert Null zwischen den Pixeln der Eingabe ergänzt und es wird ein Zero-Padding hinzugefügt. Im Anschluss wird eine Faltung durchgeführt, die das Zwischenbild als Eingabe verwendet. Die Schrittweite der Faltung beträgt hierbei immer eins. Bei einer Schrittweite von eins wird das Zwischenbild nur um die Breite des Filtermusters verkleinert, sodass das Ausgabebild größer als die Eingabe ist. Durch das Filtermuster wird ein gewichteter Mittelwert aus benachbarten Pixeln gebildet. Im Ausgabebild besitzen damit alle Pixel einen neuen Wert. Abbildung 2.12 verdeutlicht den Vorgang der transponierten Faltung.

Die Größe des Ausgabebilds einer transponierten Faltung kann mithilfe der folgenden Gleichung berechnet werden:

$$\text{Outputsize} = (\text{Inputsize} - 1) * \text{Stride} - 2 * \text{Padding} + (\text{Kernel size} - 1) + 1.$$

Es muss beachtet werden, dass der Parameter „Stride“ bei transponierten Faltungen eine andere Bedeutung hat, als bei der Faltung. Bei der normalen Faltungsoperation ist der Parameter Stride die Schrittweite der Faltung. Bei der transponierten Faltung

gibt der Parameter den Abstand zwischen zwei Eingabepixeln und damit die Anzahl der eingefügten Pixel im Zwischenbild an. Die genannte Gleichung und die damit beschriebene Vorgehensweise funktioniert jedoch nur für quadratische Datenfelder. Die Ein- und Ausgabegröße beschreibt daher sowohl Höhe, als auch Breite des Datenfelds. Diese Einschränkung ist in der vorliegenden Studienarbeit nicht problematisch. Denn die echten und die synthetischen Bilder sind quadratisch.

## 2.4. Synthetische Trainingsdaten mit GANs generieren

Bei der Recherche zu möglichen Ansätzen für das Erzeugen synthetischer Trainingsdaten mithilfe von GANs zeigte sich, dass bereits Untersuchungen existieren, die sich mit diesem Thema beschäftigen. So wurde zum Beispiel von Hung Ba untersucht, ob die Entdeckung von Kreditkartenbetrug durch den Einsatz von GANs verbessert werden kann [42]. Dabei wurden unterschiedliche GAN-Architekturen verglichen, indem synthetische Trainingsdaten mit den unterschiedlichen GANs erzeugt und im Anschluss für das Training von Klassifizierungs-KNNs verwendet wurden. Es wurde festgestellt, dass sich ein GAN dazu eignen kann, die Erkennung von Kreditkartenbetrug zu verbessern [42]. Mit keinem der Klassifizierungs-KNNs, die mit synthetischen Trainingsdaten trainiert wurden, konnte eine so hohe Genauigkeit erreicht werden, wie mit den auf echten Daten trainierten Klassifizierungs-KNNs. Jedoch konnte ein besserer F1-Wert erreicht werden [42]. In der Arbeit wird daraus der Schluss gezogen, dass mithilfe der synthetischen Daten ein ausgeglicheneres Modell trainiert wurde [42]. Trainingsdaten für die Erkennung von Kreditkartenbetrug sind oft unausgeglichen, da nur wenige Daten für den Betrugsfall existieren und korrekte Abbuchungen einen großen Anteil der Daten ausmachen. Diesem Ungleichgewicht in den Trainingsdaten kann mithilfe der synthetischen Trainingsdaten somit entgegen gewirkt werden [42].

Ähnliche Ergebnisse konnten in der Arbeit *Data Augmentation Using GANs* festgestellt werden, bei der synthetische Trainingsdaten für drei verschiedene Problemstellungen erstellt und verglichen wurden [43]. Neben einem Datensatz für Kreditkartenbetrug wurde in dieser Arbeit ein Trainingsdatensatz für die Erkennung von Brustkrebs sowie für die Diagnose von Diabetes erstellt. Die Ergebnisse ähneln denen von Hung Ba. Ein mit synthetischen Daten trainiertes Klassifizierungs-KNN für die Diagnose von Diabetes er-

reichte in der Arbeit eine etwas niedrigere Vorhersage-Genauigkeit als das mit den echten Daten trainierte Klassifizierungs-KNNs [43]. Da der Unterschied in der Genauigkeit nur minimal ist und ein besserer Recall-Wert erzielt wurde, kann auf eine ähnliche Qualität des Modells geschlossen werden [43]. Bei der Erkennung von Brustkrebs konnte sogar eine bessere Genauigkeit im Vergleich zu den echten Daten erreicht werden und auch der Recall- und Precision-Wert war höher als bei den echten Daten [43]. In der Arbeit wird der Schluss gezogen, dass GANs für die Data Augmentation geeignet sind.

Die Ergebnisse in den zwei vorgestellten Arbeiten lassen sich jedoch nur bedingt auf die vorliegende Studienarbeit übertragen, da in den aufgeführten Arbeiten nur numerische Trainingsdaten betrachtet wurden. Dennoch zeigen die Arbeiten, dass mit GANs generierte Daten prinzipiell für das Training von Klassifizierungs-KNNs geeignet sind.

Der Einsatz von synthetischen Bildern als Trainingsdaten wurde in der Arbeit von Eilertsen et al. untersucht [44]. In der Arbeit wurden mehrere GANs trainiert, um die synthetischen Trainingsdaten zu erzeugen. Die Idee dahinter ist, dass die GANs sich durch Zufall während der Optimierung auf unterschiedliche Teile der Verteilung der echten Daten fokussieren und so eine noch größere Varianz in den Daten erreicht wird [44]. Dabei wurde festgestellt, dass der Einsatz von mehreren GANs die Vorhersage-Genauigkeit eines Klassifizierungs-KNN erheblich verbessern kann [44]. Trotz des Einsatzes mehrerer GANs konnten Eilertsen et al. mit den synthetischen Trainingsdaten nicht die Performance von auf echten Daten trainierten Klassifizierungs-KNNs erreichen [44].

Im Bereich Data Augmentation hingegen konnte durch den Einsatz synthetischer Daten die Leistung von Modellen verbessert werden. Durch eine Kombination aus echten und synthetischen Daten ist eine Leistungssteigerung für Klassifizierungs-KNNs möglich. Durch den zusätzlichen Einsatz von synthetischen Bildern konnte eine um sieben Prozent bessere Performance bei der Klassifikation von Leberläsionen erreicht werden [45]. In einer weiteren Arbeit konnte mithilfe von zusätzlichen synthetischen Daten die Klassifizierung von Hautläsionen verbessert werden [46].

Das Training von Klassifizierungs-KNNs ausschließlich mit synthetischen Daten ist schwierig, da die Qualität der Daten oft nicht ausreicht, wenn z. B. nicht genug klassifizierungsrelevante Merkmale in den Daten enthalten sind. Für Data Augmentation hingegen können GANs geeignet sein und zur Leistungssteigerung eines Klassifizierungs-KNN beitragen.

## 2.5. Die C-NMC\_Leukemia-Daten

Für die vorliegende Studienarbeit werden Beispieldaten zur Klassifizierung von Leukämie Zellen verwendet [47]. Der Datensatz wird auf der Plattform Kaggle frei zur Verfügung gestellt, was eine einfache Verwendung des Datensatz ermöglicht. Der Datensatz enthält einzelne Zellbilder, die aus Mikroskopbildern segmentiert wurden. Insgesamt sind 15.135 Bilder von 118 Patienten enthalten [47]. Die Bilder sind in zwei Klassen unterteilt: Gesunde Zellen und kranke Zellen.

Die Klassifizierung als gesunde oder kranke Zelle ist eine schwierige Aufgabe, da sich die Zellen nur durch kleine Unterschiede in der Form unterscheiden. Kranke und gesunde Zellen besitzen beide die gleiche Farbe und Oberflächenstruktur. Anderer Bilddaten (z.B. von Hunden und Katze) weisen mehr klassifizierungsrelevante Merkmale auf, was ein Klassifizierung einfacher gestaltet. Für die eher schwierige Aufgabe der Zell-Klassifikation muss daher eine möglichst große Menge unterschiedlicher Bilddaten vorhanden sein, um die wenigen relevanten Merkmale richtig beurteilen zu können.



Abbildung 2.13.: Beispiele aus den C-NMC\_Leukemia-Daten:

Auf linken Seite wird eine kranke und auf der rechten Seite eine gesunde Zelle abgebildet [47].

In Abbildung 2.13 ist zu erkennen, dass die Zellen in den Bilddateien von einem relativ großen schwarzen Bereich umgeben sind. Dieser Bereich ist auf allen Bildern zu finden und enthält keine klassifizierungsrelevanten Informationen. Daher werden die Bilder von den ursprünglichen 450x450 Pixeln auf eine Größe von 300x300 Pixeln zugeschnitten. Die Verarbeitung von kleineren Bildern erfordert weniger trainierbare Parameter (z. B. Gewichte), wodurch das Training effizienter gestaltet wird.

## 3. Konzepte der Studienarbeit

In diesem Kapitel werden verschiedene GANs und Klassifizierungs-KNNs konzipiert. Die konzipierten Modelle werden in den anschließenden Kapiteln dazu verwendet, um das Potential synthetischer Bilddaten zur Optimierung des Trainings von Klassifizierungs-KNNs festzustellen.

In Unterkapitel 3.1 werden drei Eigenschaften von Neuronalen Netzen beschrieben, die bei der Betrachtung von Konzepten für GANs und Klassifizierungs-KNNs beachtet werden müssen. In Unterkapitel 3.2 werden Konzepte für GANs zum Erzeugen synthetischer Bilddaten anhand vorgegebener Labels beschrieben. Im anschließenden Unterkapitel 3.3 werden verschiedene Konzepte von Klassifizierungs-KNNs vorgestellt. Die grundlegenden Konzepte werden durch die Wahl von Hyperparametern wie z. B. der Aktivierungs- und Loss-Funktion konkretisiert.

### 3.1. Design-Aspekte von KNNs

Bei der Beschreibung der Konzepte für GANs und Klassifizierungs-KNNs muss besonders auf die folgenden drei Eigenschaften von KNNs geachtet werden. Anhand dieser Eigenschaften können KNNs miteinander verglichen werden.

#### 1. Größe eines KNN-Modells (Kosten):

Für das Training werden KNNs in den Arbeitsspeicher der Laufzeit bzw. den Video Random Access Memory (VRAM) einer Grafikkarte geladen. Dieser Speicher ist begrenzt und wird zusätzlich zu dem KNN auch durch die Berechnungen während des Trainings teilweise belegt. Wenn der Speicher dafür nicht ausreicht, kann ein KNN nicht trainiert werden. Aus diesem Grund muss darauf geachtet werden, dass die Anzahl von Neuronen und Gewichten der KNNs möglichst klein ist.

#### 2. Leistungsfähigkeit eines KNN-Modells (Qualität):

Die Leistungsfähigkeit eines Klassifizierungs-KNN wird i. d. R. an der Vorhersage-

Genauigkeit und der Abweichung der Vorhersagen von den tatsächlichen Ergebnissen (Loss-Wert) gemessen.

Die Leistungsfähigkeit eines GAN-Modells zu bewerten ist schwieriger. Denn die Beschaffenheit der Ergebnisse von GANs erschweren die Definition einer passenden Metrik. Der menschliche Entwickler des GAN ist dagegen in der Lage, die Qualität generierter Bilder zumindest grob einzuschätzen. Die menschliche Beurteilung ist allerdings subjektiv und verschiedene Personen können ein Ergebnis unterschiedlich beurteilen. Zudem können Menschen bei großen Datensätzen nur schwierig die Generalisierungsfähigkeit eines GAN einschätzen und übersehen ein Overfitting. Trotz der Probleme ist die menschliche Beurteilung während der Entwicklung von Vorteil. So kann z. B. überprüft werden, ob das GAN Lernfortschritte erzielt oder es zu einem Mode Collapse kommt.

Zusätzlich zu der menschlichen Beurteilung wird die Fréchet Inception Distance (FID) verwendet, um das GAN-Modell zu beurteilen. Die FID ist eine Metrik um die Ähnlichkeit zwischen den Bildern aus zwei Datensätzen zu bewerten [48]. In der Praxis hat sich gezeigt, dass der FID-Score oft mit der menschlichen Beurteilung übereinstimmt [48]. Die FID wird in der vorliegenden Studienarbeit verwendet, da sie einer der verbreitetsten Ansätze zur Bewertung der Leistungsfähigkeit von GANs ist [48].

### 3. Trainingsdauer eines KNN-Modells (Zeit):

Das Training von KNNs ist nicht nur speicher- und rechenaufwändig, sondern auch sehr zeitaufwändig. Durch die Topologie und Hyperparameter eines KNN-Modells wird seine Trainingsdauer beeinflusst. Um den Zeitrahmen für die vorliegende Studienarbeit einhalten zu können, muss ein KNN-Konzept gewählt werden, das nach einer annehmbaren Trainingsdauer gute Ergebnisse liefert.

## 3.2. Konzept für die GANs

Ziel dieses Unterkapitels ist es, leistungsstarke GANs für das Erzeugen synthetischer Trainingsdaten zu entwickeln. Dafür werden aktuelle Ansätze und Techniken im Hinblick auf das Ziel der Studienarbeit bewertet und verglichen. Zudem werden die verwendeten

Techniken, Architekturen und Schichten beschrieben. Für die Bewertung der vorgestellten Konzepte werden die oben beschriebenen Kriterien herangezogen. Am Ende sollen mit den GANs den Trainingsdaten möglichst ähnliche Bilder generiert werden.

Der Ausgangspunkt für die Entwicklung von GANs für die vorliegende Studienarbeit ist die CGAN-Architektur. Diese Architektur wird gewählt, da es möglich sein soll, gezielt Bilder für ein gegebenes Label zu erzeugen.

Eine andere Möglichkeit, um zielgerichtet Trainingsdaten zu generieren, ist für jede Klasse ein eigenes GAN zu erstellen. Ob diese Variante zu besseren synthetischen Daten führt, lässt sich nicht ohne Weiteres sagen. Die in der vorliegenden Studienarbeit verwendeten Trainingsdaten unterscheiden nur zwischen zwei Klassen. Der Rechenaufwand für separate GANs für jede Klasse ist daher noch vertretbar. Im Allgemeinen scheint dieser Ansatz allerdings nicht praktikabel, da die Anzahl der GANs direkt proportional zur Anzahl der Problemklassen ist. Daher wurde sich für die Variante mit einem CGAN entschieden.

## Der Generator

Beim Generator wird als Eingabeschicht eine vollverknüpfte Schicht mit 300 Neuronen verwendet. Die Größe von 300 Neuronen wurde gewählt, um im Anschluss aus den 300 Ausgabewerten drei Bilder mit jeweils 10x10 Pixeln zu erzeugen. Die Idee dahinter ist, dass mit den drei Bildern die Farbkanäle des RGB-Farbraums repräsentiert werden.

Insgesamt erwartet die Eingabeschicht 130 Parameter, wovon 128 zufällig generiert sind und die anderen beiden für das Label (One-Hot-Kodierung) verwendet werden. Die 128 zufälligen Startwerte entstammen einer Standardnormalverteilung, da durch diese Initialisierung bessere Ergebnisse zu erwarten sind [6]. Es wurde sich für 128 Startwerte entschieden, da diese Anzahl in der Praxis etabliert ist [6]. Alternativ werden in der Praxis häufig 100 zufällige Startwerte genutzt, aufgrund der höheren Anzahl an Neuronen in der Eingabeschicht wurde sich aber für 128 Eingabewerte entschieden. Mit diesem Aufbau in der Eingabeschicht wird dem klassischen Aufbau einer CGAN Architektur gefolgt. Im Anschluss an die Eingabeschicht folgen drei versteckte Schichten.

Die versteckten Schichten des Generators sind alle gleich aufgebaut und bestehen aus transponierten Faltungsschichten mit anschließender Batch-Normalisierung (siehe Abbildung 3.1). Die Normalisierung der Kanäle führt meistens zu einem besseren Ergebnis [6].

Als Aktivierungsfunktion wird in allen versteckten Schichten die Leaky ReLU Funktion verwendet. Die Leaky ReLU Funktion wird verwendet, um dem Problem der verschwindenden Gradienten vorzubeugen.

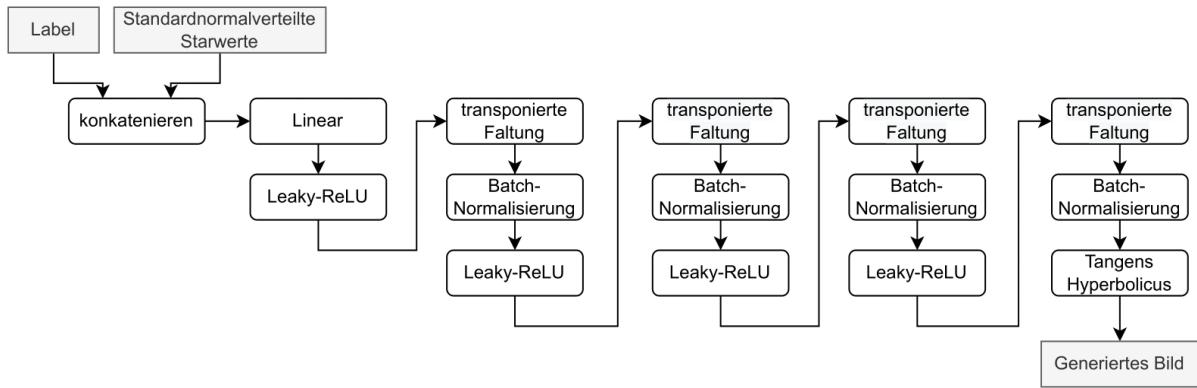


Abbildung 3.1.: Architektur des Generators:

Zuerst werden die zufällig standardnormalverteilten Startwerte mit dem Label konkateniert. Im Anschluss wird mit einer vollverknüpften Schicht die benötigte Anzahl an Pixelwerten erzeugt. Mit mehreren transponierten Faltungsschichten inklusive Batch-Normalisierung und Leaky ReLU Aktivierungsfunktion werden aus diesen Zufallswerten größere Datenfelder erzeugt. Am Ende wird mithilfe der Tangens Hyperbolicus Funktion der Wertebereich der Ausgabe beschränkt, sodass eine Interpretation der Ausgabe als Bild möglich ist.

Im Anschluss an die drei versteckten Schichten folgt eine Ausgabeschicht, die ähnlich aufgebaut ist wie die versteckten Schichten. Die Ausgabeschicht verwendet eine transponierte Faltungsschicht mit anschließender Batch-Normalisierung. Anstatt der Leaky ReLU Funktion wird der Tangens Hyperbolicus als Aktivierungsfunktion verwendet. Eine Alternative zum Tangens Hyperbolicus ist die Sigmoid Funktion. Beide Funktionen haben unterschiedliche Vor- und Nachteile. In der Praxis hat sich der Tangens Hyperbolicus als Aktivierungsfunktion für die Ausgabeschicht in GANs etabliert. Deshalb wird diese Funktion auch hier benutzt.

Für ein abgeschlossenes Konzept für den Generator müssen für alle transponierten Faltungsschichten Parameter wie z. B. die Größe der Filtermuster festgelegt werden. Dieser Schritt ist eine große Herausforderung, da die Auswahl der Parameter die Leistung des Generators stark beeinflusst. Besitzt der Generator aufgrund der gewählten Werte zu

viele Freiheiten, kann es zum Overfitting kommen. Ist das Modell zu stark beschränkt, kann die Komplexität eines Problems möglicherweise nicht vollständig erfasst werden.

Das Finden geeigneter Parameter ist ein zeitaufwändiger Prozess. Konkret mussten Entscheidungen für die folgenden vier Parameter getroffen werden:

- Anzahl der Eingabekanäle
- Anzahl der Ausgabekanäle
- Größe der Filtermuster
- Stride

In der Eingabeschicht wird aus den Eingabewerten ein  $3 \times 10 \times 10$  großes Datenfeld erzeugt, das ein  $10 \times 10$  RGB-Bild repräsentiert. Daraus soll ein  $3 \times 300 \times 300$  großes Bild erzeugt werden. Um das zu erreichen, muss die Größe der Filtermuster und der Stride für alle vier transponierten Faltungsschichten passend ausgewählt werden. Das wird durch die in den Grundlagen eingeführten Gleichung zur Berechnung der Ausgabegröße einer transponierten Faltung vereinfacht. Der Stride-Parameter sollte dabei möglichst klein gewählt werden, damit viele der benachbarten Pixel in die Berechnung des Farbwertes einfließen. Im besten Fall beträgt der Wert des Stride-Parameters zwei, da so genau ein leerer Pixel zwischen den bestehenden Pixeln eingefügt wird. Für die Größe der Filtermuster werden Werte kleiner zehn akzeptiert. Diese Beschränkung erscheint sinnvoll, da sich in diesem Bereich vermutlich der Großteil der relevanten Nachbar-Pixel befindet. Zudem verlangsamten große Filtermuster das Training. Auf Grundlage dieser Überlegungen wurde die folgenden Parameter-Kombinationen festgelegt:

1. transponierte Faltung: Stride=2, Kernelsize=4  $\Rightarrow (10 - 1) * 2 - 2 * 0 + (4 - 1) + 1 = 22$
2. transponierte Faltung: Stride=2, Kernelsize=6  $\Rightarrow (22 - 1) * 2 - 2 * 0 + (6 - 1) + 1 = 48$
3. transponierte Faltung: Stride=2, Kernelsize=6  $\Rightarrow (48 - 1) * 2 - 2 * 0 + (6 - 1) + 1 = 100$
4. transponierte Faltung: Stride=3, Kernelsize=3  $\Rightarrow (100 - 1) * 3 - 2 * 0 + (3 - 1) + 1 = 300$

Die Anzahl der Eingabekanäle in eine transponierte Faltungsschicht muss mit der Anzahl der Ausgabekanäle der vorherigen transponierten Faltungsschicht übereinstimmen. Die Anzahl der Ausgabekanäle kann hingegen frei gewählt werden. Dieser Parameter ist ele-

mentar, da er den größten Einfluss auf die Anzahl der trainierbaren Parameter und damit auf die Leistung und Trainingsdauer des Generators hat. Üblicherweise werden sowohl bei der Faltung als auch bei der transponierten Faltung Zweierpotenzen für die Anzahl der Kanäle verwendet. Dadurch gibt es nur noch wenige Möglichkeiten für die Auswahl des Parameter-Werts. Vergleichbare Generatoren aus der Literatur verwenden i. d. R. Werte größer als  $2^8$  und kleiner als  $2^{13}$ . Ausgehend von diesen Grenzwerten wurde die folgende Auswahl für die Ein- und Ausgabekanäle der transponierten Faltungsschichten des Generators getroffen:

1. transponierte Faltung: Eingabekanäle=3, Ausgabekanäle=512
2. transponierte Faltung: Eingabekanäle=512, Ausgabekanäle=1024
3. transponierte Faltung: Eingabekanäle=1024, Ausgabekanäle=1024
4. transponierte Faltung: Eingabekanäle=1024, Ausgabekanäle=3

Die Trainingsdauer war für die gewählte Anzahl an Kanälen akzeptabel, da bereits nach relativ kurzer Zeit Trainingsfortschritte erkannt werden konnten. Nach einer kurzen Trainingsdauer wurden bereits Bilder generiert, die den echten Bildern ähneln und unterschiedliche Zellen darstellen. Es war kein Mode Collapse erkennbar. Aus diesen Beobachtungen wurde der Schluss gezogen, dass die Anzahl der trainierbaren Parameter ausreichend ist, um das betrachtete Problem zu modellieren. Die Werte für Stride, Kernelsize und Anzahl der Ausgabekanäle werden in Abbildung 3.2 zusammengefasst.

Mit einer Größe von drei versteckten Schichten kann der Aspekt der Modell-Größe für den Generator als erfüllt angesehen werden. Der VRAM ist für das Modell mehr als ausreichend. Eine separate Betrachtung des Generators wirft im Hinblick auf die Trainingsdauer keine Probleme auf. Jedoch besteht das Training eines GAN aus drei Schritten und bezieht dabei den Diskriminator mit ein. Der Generator muss je Trainingsschritt drei Mal vorwärts und einmal rückwärts durchlaufen werden. Dadurch ist beim Training von GANs vor allem die Trainingsdauer der limitierende Faktor. In ersten Versuchen zeigte sich, dass die Trainingsdauer je nach Parameter-Auswahl für einen einzigen Trainingsschritt bereits 20-30 Minuten betrug. Es muss beachtet werden, dass dabei der Diskriminator mit inbegriffen ist. Das ist die maximal akzeptable Trainingsdauer. Zusätzlichen Schichten im Generator erhöhen die Trainingsdauer und sind daher ausgeschlossen.

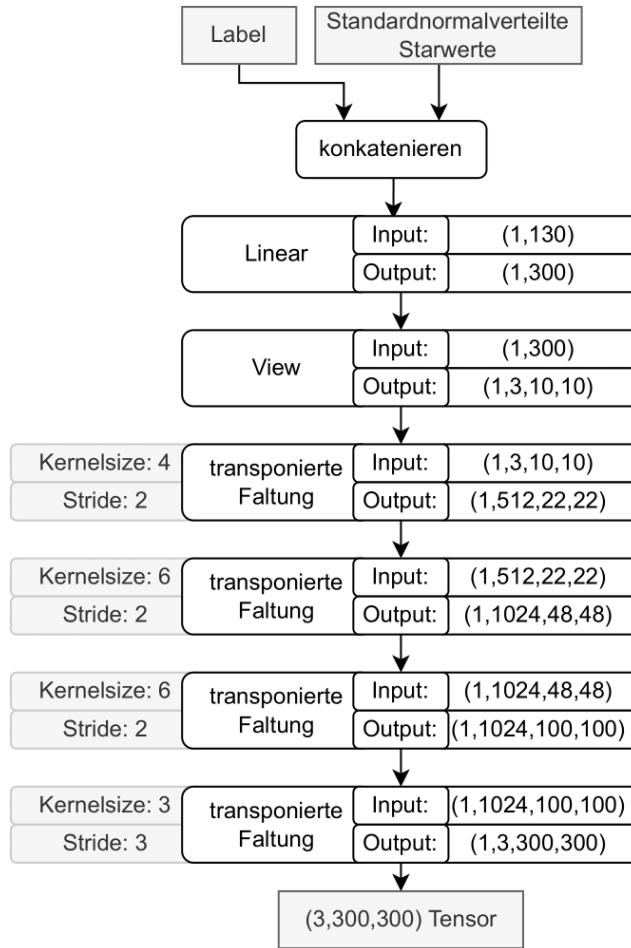


Abbildung 3.2.: Parameter des Generator-Netzwerks: Durch die gewählten Parameter wird das eingehende Datenfeld mit jeder transponierten Faltungsschicht vergrößert, sodass der Ausgabetensor der letzten transponierten Faltungsschicht ein RGB-Bild repräsentiert.

## Der Diskriminatator

Beim Diskriminatator werden Faltungsschichten genutzt, um die Bilddaten effizient zu verarbeiten. Bei der Klassifizierung von Bilddaten werden oft Netze verwendet, die ausschließlich aus Faltungsschichten bestehen, da diese Netze als besonders leistungsstark gelten [6]. Deshalb verwendet der Diskriminatator bis auf die letzte Schicht nur Faltungsschichten. Die letzte Schicht besteht aus einem einzigen Neuron, das die Wahrscheinlichkeit ausgibt, ob es sich um ein echtes oder ein generiertes Bild handelt. Damit die Ausga-

be ein Wahrscheinlichkeitswert ist, muss die Ausgabe auf das Intervall  $[0,1]$  beschränkt werden. Deshalb wird für das Neuron die Sigmoid-Aktivierungsfunktion genutzt.

Auf die Eingabeschicht folgen beim Diskriminator zwei versteckte Schichten, wobei die Eingabeschicht und die versteckten Schichten gleich aufgebaut sind. Jede der Schichten besteht aus einer Faltungsschicht mit anschließender Batch-Normalisierung. Als Aktivierungsfunktion wird die Leaky ReLU Funktion verwendet. Mit diesem Aufbau ist die Architektur des Diskriminators das Gegenstück zur Architektur des Generators. Für ein erfolgreiches Training von GANs muss die Leistung des Diskriminators und die des Generators ausgeglichen sein. Dieses Gleichgewicht geht häufig durch einen zu starken Diskriminator verloren. Um das zu verhindern, folgt auf die transponierten Faltungsschichten inklusive Batch-Normalisierung und Aktivierungsfunktion jeweils eine Dropout-Schicht. Durch diese Dropout-Schichten soll die Leistung des Diskriminators verschlechtert werden, damit Generator und Diskriminator möglichst immer im Gleichgewicht sind. Abbildung 3.3 verdeutlicht die Struktur des Diskriminators.

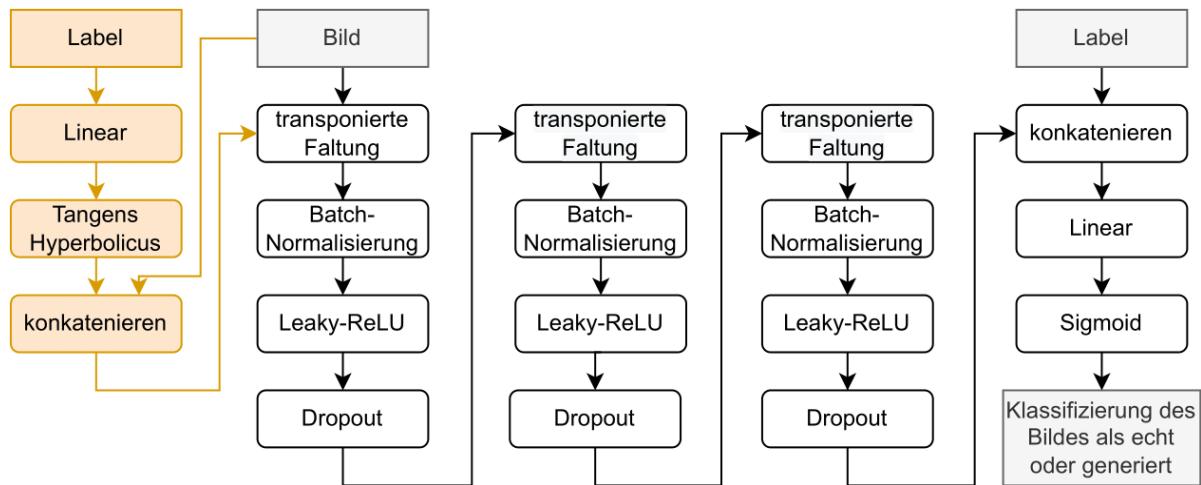


Abbildung 3.3.: Architektur des Diskriminators:

Für die Klassifizierung eines Bilds als echt oder synthetisch werden nur Faltungsschichten verwendet. Erst vor der Ausgabeschicht wird das Label mit den der Ausgabe der letzten Faltungsschicht konkateniert. In Orange ist eine alternative Variante für die Verarbeitung des Labels dargestellt. Bei dieser Variante wird das Label mit einer vollverknüpften Schicht auf die Größe der Bilder skaliert und als zusätzlicher Kanal mit in die Faltung gegeben.

Damit der Diskriminator auch das One-Hot-codierte Label berücksichtigt, wird das Label als zusätzliche Eingabe in die letzte Schicht des Diskriminators gegeben [49]. Die Eingabe der letzten Schicht besteht somit aus den zwei Werten des One-Hot-codierten Labels sowie den Merkmalskarten der letzten Faltungsschicht (siehe Abbildung 3.3). Alternativ kann das Label in Form eines zusätzlichen Kanals eines Bildes in die erste Faltungsschicht eingespeist werden (siehe Abbildung 3.3) [50]. Bei dieser Variante wird in der Regel eine vollverknüpfte Schicht genutzt, um aus dem codierten Label ein oder mehrere Bilder zu erzeugen [50]. Welche der beiden Varianten vielversprechender ist, kann nicht im Voraus gesagt werden, da in diese Richtung nur wenige Untersuchungen existieren.

Das Label in der letzten Schicht des Diskriminators zu berücksichtigen, scheint der ursprünglichen CGAN Architektur näher zu kommen. Bei der ursprünglichen CGAN Architektur wurden ebenfalls vollverknüpfte Schichten verwendet, um ein Label und die Eingabe zu verarbeiten. Des Weiteren sind Faltungsschichten für die Extraktion von Merkmalen gedacht. Die Extraktion von Merkmalen aus den „Label-Bildern“ erscheint nicht sinnvoll. In ersten Versuchen zeigte sich schnell, dass der zweite Ansatz nicht erfolgreich ist. Wenn das Label als zusätzlicher Bildkanal der Eingabe hinzugefügt wird, kommt es nach wenigen Trainings-Epochen zum Mode Collapse. Abbildung 3.4 zeigt die Ausgabe und den Mode Collapse des Generators nach zwei Trainings-Epochen.

Aufgrund des Mode Collapse wurden verschiedene potentielle Verbesserungen für die Transformation des Labels in den Bildkanal getestet. Als Aktivierungsfunktion wurden neben dem Tangens Hyperbolicus die ReLU und die Leaky ReLU Funktion getestet. Jedoch konnte der Mode Collapse damit nicht verhindert werden. Das Weglassen einer Aktivierungsfunktion führte ebenfalls nicht zum Erfolg. Eine Normalisierung der Werte war ebenso erfolglos. Mit keinem Ansatz konnte der Mode Collapse verhindert werden. Deshalb wurde der Ansatz verworfen. Stattdessen wird dem Diskriminator das Label erst nach der letzten Faltungsschicht übergeben. Dieser Ansatz stellt sich als geeignet heraus, da nie ein Mode Collapse festgestellt wurde. Dieser Schluss ist auf die verwendete Architektur beschränkt, da bei anderen Projekten beide Ansätze erfolgreich sind [49, 50].

Für die Umsetzung des Diskriminators müssen die gleichen Parameter wie schon beim Generator festgelegt werden. Die Größe der Ausgabe der letzten Faltungsschicht ist beim Diskriminator nicht fest vorgegeben. Die einzige Anforderung ist, dass die Faltungsschichten die Eingabe so weit verkleinern müssen, dass die Merkmalskarten im Anschluss durch

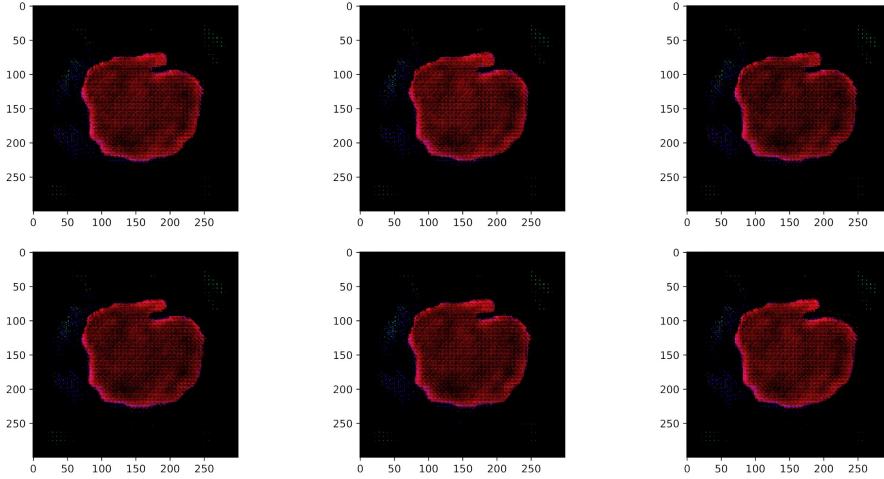


Abbildung 3.4.: Der Mode Collapse des Generators nach zwei Trainings-Epochen:  
Wenn das Label als zusätzlicher Kanal in den Diskriminatoren eingespeist wird, kommt es schon nach zwei Trainings-Epochen zum Mode Collapse.

eine vollverknüpfte Schicht weiterverarbeitet werden können. Wenn die Merkmalskarten nach der letzten Faltungsschicht zu groß sind, benötigt die anschließende vollverknüpfte Schicht aufgrund der großen Anzahl an Neuronen und Parametern zu viel Speicherplatz und Zeit für das Training.

Um passende Werte für die Größe der Filtermuster und den Stride der Faltungsschichten zu finden, wird die in Unterkapitel 2.2 eingeführte Gleichung zur Berechnung der Ausgabegröße einer Faltungsschicht verwendet. Für den Stride-Parameter wird immer der Wert Zwei genutzt, da dieser Wert in der Praxis stark verbreitet ist und gleichzeitig eine effektive Reduktion der Größe der Merkmalskarten innerhalb weniger Schichten ermöglicht. Die Größe der Filtermuster wird so gewählt, dass keine Randpixel während der Faltung ignoriert werden. Die folgenden Werte für die Größe der Filtermuster und den Stride erwiesen sich als geeignet:

1. Faltung: Stride=2, KernelSize=8  $\Rightarrow \frac{300-8}{2} + 1 = 147$
2. Faltung: Stride=2, KernelSize=9  $\Rightarrow \frac{147-9}{2} + 1 = 70$

$$3. \text{ Faltung: Stride}=2, \text{ KernelSize}=8 \Rightarrow \frac{70-8}{2} + 1 = 32$$

Mit einer Größe von 32x32 Pixeln sind die Merkmalskarten nach der dritten Faltung ausreichend klein, um im Anschluss mit vollverknüpften Schichten arbeiten zu können. Für die Suche nach einer passenden Anzahl an Ein- und Ausgabekanälen wurden wie beim Generator Zweierpotenzen verwendet, wobei sich wieder an der Auswahl vergleichbarer Diskriminatoren orientiert wurde. Abbildung 3.5 fasst die Konfiguration der Größen der Filtermuster und der Strides zusammen.

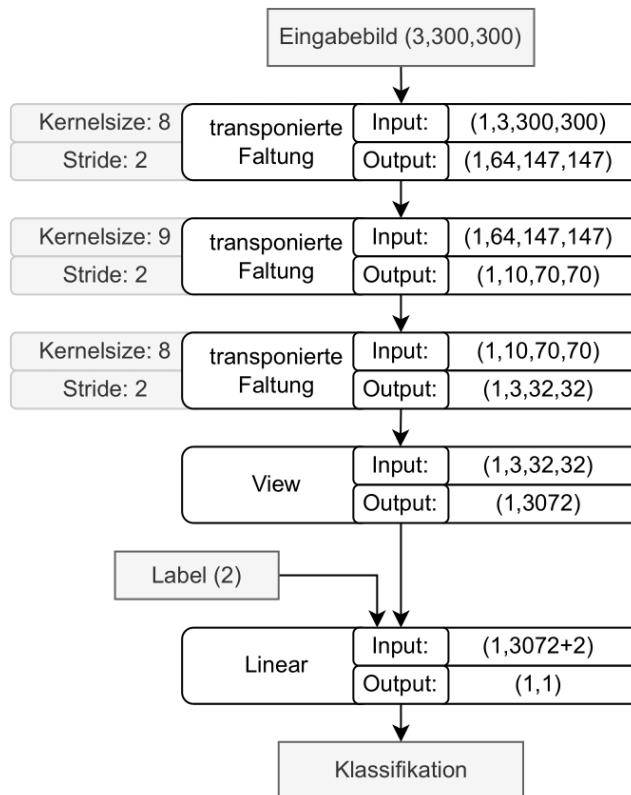


Abbildung 3.5.: Parameter des Diskriminator-Netzwerks:

Mithilfe von Faltungsschichten werden aus dem Eingabetensor die Bildmerkmale extrahiert. Die Merkmalskarten der letzten Faltungsschicht werden zusammen mit dem Label in eine vollverknüpfte Schicht überführt. Die vollverknüpfte Schicht besitzt ein Neuron, das die Echtheit eines Bilds bestimmt.

Für die Dropout Schichten muss der Anteil der Neuronen festgelegt werden, die während des Trainings zufällig ausgeschaltet werden. Die Leistung der GANs ist für jede verwendete Loss-Funktion unterschiedlich. Deshalb muss der prozentuale Anteil deaktivierter

Neuronen für jede Loss-Funktion individuell bestimmt werden. Die Kriterien für die Auswahl einer Loss-Funktion sind der Trainingsfortschritt von Diskriminator und Generator sowie die Qualität der generierten Bilder.

Nach dem Training der GANs für einige Epochen, werden die Trainingskurven von Diskriminator und Generator sowie die erzeugten Bilder betrachtet. Wenn die Änderung des Loss-Werts in den Lernkurven und die erzeugten Bilder einen Traingsfortschritt nahelegen, befinden sich Generator und Diskriminator im Gleichgewicht. Gestartet wurde die Suche nach dem optimalen Anteil deaktivierter Neuronen bei null Prozent. Der Anteil wurde in 10% Schritten erhöht. Nach einigen Versuchen stellten sich für die unterschiedlichen Loss-Funktionen folgende Werte als geeignet heraus:

- Binäre Kreuzentropie: 40%
- Quadratischer Fehler: 20%
- Wasserstein-Distanz: Es sind keine Dropout Schichten nötig, da ein leistungsstarker Diskriminator hier von Vorteil ist (siehe unten).

Die Größe des Diskriminators ist wie schon beim Generator nicht der beschränkende Faktor. Der zur Verfügung stehende VRAM ist für den Diskriminator alleine und in Kombination mit dem Generator ausreichend. Da der Diskriminator pro Trainingsschritt drei Mal vorwärts und zwei mal rückwärts durchlaufen werden muss, hat er maßgeblichen Einfluss auf die Trainingsdauer eines GAN. Aus diesem Grund ist der Diskriminator mit zwei versteckten Schichten etwas kleiner als der Generator. Im Hinblick auf die Trainingsdauer ist mit dem vorgestellten Konzept das Maximum erreicht. Die Leistungsfähigkeit scheint ebenfalls ausreichend zu sein. Diese musste sowohl bei Verwendung der binären Kreuzentropie als auch des quadratischen Fehlers durch Dropout-Schichten reduziert werden, um ein Gleichgewicht zwischen Diskriminator und Generator zu erreichen.

Bei Verwendung der Wasserstein Distanz kann die Leistung des Diskriminators erhöht werden, indem dieser öfter trainiert wird als der Generator. Dieses Vorgehen, um die Leistung des Diskriminators bei Verwendung der Wasserstein-Distanz zu erhöhen, entspricht dem empfohlenen Ansatz [40, 41].

## Die Loss-Funktion

Die Wahl einer einzigen Loss-Funktion scheint nicht sinnvoll, da die Leistung eines GAN stark von der Loss-Funktion abhängt. Aus diesem Grund werden GANs mit den folgenden Loss-Funktionen implementiert und verglichen:

- Binäre Kreuzentropie (BCE),
- Quadratischer Fehler,
- Wasserstein-Distanz.

Die binäre Kreuzentropie wurde bereits von Ian Goodfellow verwendet und ist damit die klassische Fehlerfunktion für GANs. Diese Loss-Funktion ist eine naheliegende Wahl, da der Diskriminatator eine Wahrscheinlichkeitsverteilung ausgibt und die Kreuzentropie genau für diesen Fall konzipiert ist. Der quadratische Fehler gehört ebenfalls zu den verbreiteten Loss-Funktionen. Mit dieser Loss-Funktion konnten in einigen Fällen sogar bessere Ergebnisse erzielt werden, als mit der binären Kreuzentropie [51].

Die Wasserstein-Distanz ist eine der aktuell vielversprechendsten Fehlerfunktion für GANs und soll deshalb ebenfalls implementiert und getestet werden [40, 41]. In der Architektur des GAN wird durch die Wahl der Wasserstein-Distanz eine Anpassung nötig. Sowohl bei der binären Kreuzentropie als auch beim quadratischen Fehler wird die Ausgabe des Diskriminators als Wahrscheinlichkeit interpretiert, so wie es im ursprünglichen Konzept von Ian Goodfellow gedacht war.

Diese Interpretation ist bei WGANs allerdings nicht mehr richtig, da der Diskriminatator beim WGAN keine Wahrscheinlichkeit mehr ausgibt, sondern eine Kritik [40]. Damit der Diskriminatator keine Wahrscheinlichkeit mehr liefert, sondern eine Bewertung, muss die Aktivierungsfunktion in der letzten Schicht angepasst werden. Bei den WGANs wird in der Ausgabeschicht die Leaky ReLU Funktion verwendet, die einen numerischen Wert berechnet.

Des Weiteren muss sich die Leistung von Generator und Diskriminatator bei Verwendung der Wasserstein-Distanz nicht mehr im Gleichgewicht befinden. Die vom Diskriminatator geäußerte Kritik darf beliebig groß sein. Aus diesem Grund können bei der Variante mit der Wasserstein-Distanz die Dropout-Schichten im Diskriminatator entfernt werden [40].

### Das Trainings-Verfahren

Das Training wird mit dem ADAM-Verfahren durchgeführt, da dieses Verfahren allgemein und auch speziell für GANs weit verbreitet ist [6]. Als Lernrate für das ADAM-Verfahren wird ein Startwert von 0.0002 verwendet. Dieser Wert hat sich als gute Wahl etabliert [6].

## 3.3. Konzept für die Klassifizierungs-KNNs

In diesem Unterkapitel werden Konzepte für Klassifizierungs-KNNs vorgestellt. Wie bereits in Unterkapitel 2.2 beschrieben, sind klassische KNNs in der Praxis nicht für die Verarbeitung von Bildern geeignet. Darum werden nachfolgend ausschließlich CNN-Konzepte beschrieben.

Leistungsstarke Klassifizierungs-KNNs zu konstruieren, kann ein langwieriger Prozess sein. Denn KNNs sind empfindliche Konstrukte, die schon bei kleinen Änderungen der Topologie oder der Hyperparameter (z. B. Lernrate) große Ergebnis-Unterschiede liefern können. Hinzu kommt, dass die Art der Problemstellung und ganz besonders die verwendeten Daten über den Trainingserfolg eines KNN entscheiden. Eine passende Konfiguration eines KNN kann daher oft nur durch Trial-and-Error gefunden werden [5, 17].

Um die Suche nach passenden Konfigurationen für Klassifizierungs-KNNs in der vorliegenden Studienarbeit abzukürzen, werden bestehende Konzepte für Klassifizierungs-KNNs herangezogen. Die Konzepte umfassen dabei vorrangig die Topologie eines KNN. Durch die Wahl von Hyperparametern werden die Konzepte auf die in der vorliegenden Studienarbeit betrachtete Problemstellung und die verwendeten Daten angepasst.

### Konzepte für Klassifizierungs-KNNs

Es gibt eine große Bandbreite an Konzepten für Klassifizierungs-KNNs. Einige der bekanntesten und erfolgreichsten dieser Konzepte für Klassifizierungs-KNNs wurden in Pytorch implementiert und können für eigene Problemstellungen adaptiert werden (siehe <https://pytorch.org/vision/stable/models.html>). Für die vorliegenden Studienarbeit

werden drei dieser in Pytorch implementierten Konzepte herangezogen. Die jeweilige Implementierung der drei Klassifizierungs-KNN-Konzepte wurden jeweils so ausgewählt, dass der Speicherbedarf und die Trainingsdauer im akzeptablen Rahmen bleibt. Nachfolgend werden diese drei Konzepte skizziert:

### 1. Inception Neural Networks (IncNets):

Das IncNet-Konzept wurde erstmals in einem „GoogLeNet“ genannten KNN verwendet und gewann die ImageNet Large Scale Visual Recognition Challenge (ILSVCR) im Jahr 2014. Die Idee hinter dem IncNet-Konzept ist, dass klassifizierungsrelevante Informationen gleichzeitig aus mehreren verschiedenen großen Bereichen extrahiert werden können. Um dies umzusetzen, verwenden IncNets neben Faltungs- und Bündelungsschichten auch „Inception Module“. Diese Inception Module bestehen aus parallel geschalteten Faltungs- und Bündelungsschichten, die verschiedene große Filtermuster und Bündelungsfelder nutzen. Die ausgehenden Merkmalskarten der parallel geschalteten Schichten werden zu der Ausgabe des Inception Moduls konkateniert. Dadurch können mit einem Inception Modul Informationen parallel aus verschiedenen Kontexten extrahiert werden. Das IncNet lernt im Trainingsprozess, welche dieser Informationen relevant für die Problemstellung sind und gewichtet die betreffenden Berechnungen entsprechend [5, 52]. Abbildung 3.6 verdeutlicht die Topologie der Inception Module.

Mit diesem Ansatz konnte GoogLeNet größere Genauigkeiten als bisherige KNNs erzielen. Und obwohl das IncNet-Konzept auf parallel geschalteten Schichten basiert, war das Training von GoogLeNet effizienter als das Training des damals stärksten Konkurrenz-KNN „VGGNet“. Durch Anpassungen wie z. B. Bottleneck-Schichten (siehe Abbildung 3.6) oder die Faktorisierung von Faltungsschichten wurde die Trainingseffizienz des IncNet-Konzepts bis heute weiter optimiert [52, 53]. Für die vorliegende Studienarbeit wird die dritte Version des IncNet-Konzepts (IncNet V3) verwendet.

### 2. Residual Neural Networks (ResNets):

Das ResNet-Konzept gewann im Jahr 2015 die ILSVCR und übertraf die Gewinner-Architektur des Vorjahres in Genauigkeit und Effizienz. Seitdem wurde das ResNet-Konzept häufig adaptiert und verbessert. ResNets bestehen hauptsächlich aus Faltungsschichten, die anstelle von Bündelungsschichten auch für die Verdichtung von Daten verwendet werden. Maßgeblich für das ResNet-Konzept sind die Shortcut-Verbindungen, mit denen Daten zwei Neuronen-Schichten überspringen können (siehe Abbildung 3.7). Diese

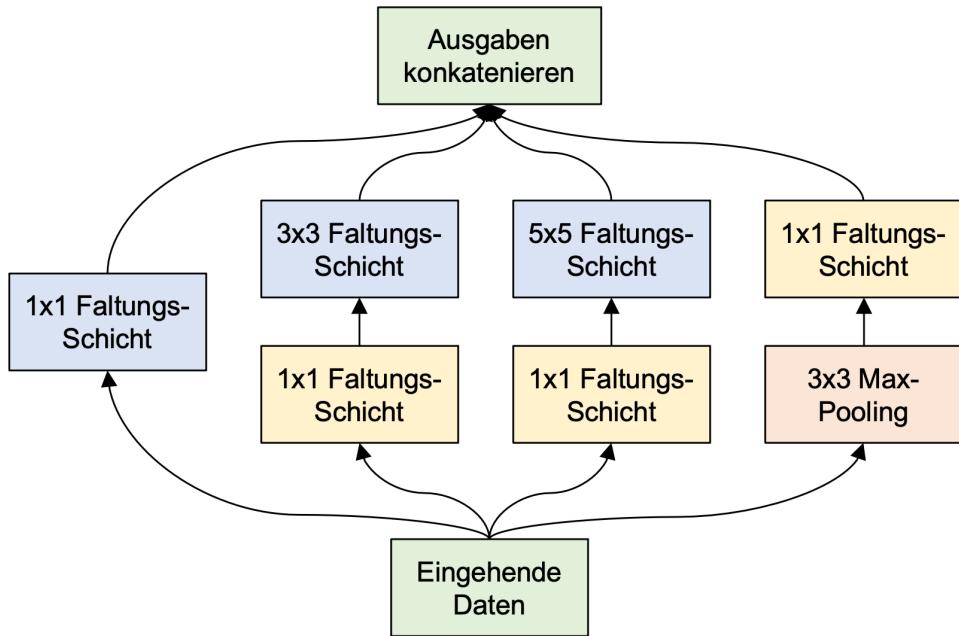


Abbildung 3.6.: Ein Inception Modul mit Bottleneck-Schichten:

Das IncNet-Konzept führt Inception Module ein, in denen mehrere unterschiedliche Schichten parallel verwendet werden. So können mehr klassifizierungsrelevante Informationen extrahiert werden, als durch eine einfache Sequenz von Schichten. Durch Bottleneck-Faltungsschichten (in Gelb) wird die Anzahl der Merkmalskarten auf 1 gesenkt, ohne eine Faltung durchzuführen ( $1 \times 1$  mit Schrittweite 1). Das reduziert den Rechenaufwand für die Faltungs- und Bündelungsschichten die das „echte“ Merkmalslernen durchführen (in Blau und Rot).

alternativen Wege ermöglichen das effiziente Training von ResNets mit vielen Schichten. Die große Anzahl versteckter Faltungsschichten, die in ResNets verwendet werden können, bergen das Potential einer hohen Vorhersage-Genauigkeit [5, 16]. Für die vorliegende Studienarbeit wird eine Implementierung des ResNet-Konzepts mit 18 Faltungsschichten (ResNet18) verwendet.

### 3. EfficientNet:

Das EfficientNet-Konzept wurde 2019 von Google vorgestellt. EfficientNets bestehen hauptsächlich aus Faltungsschichten und verwenden ähnlich wie das ResNet-Konzept Shortcut-Verbindungen. Die Besonderheit des EfficientNet-Konzepts liegt im so genannten „compound scaling“. Dabei werden Auflösung, Breite und Tiefe des KNN automatisch skaliert, um die Performance des Netzes bestmöglich zu verbessern. Andere

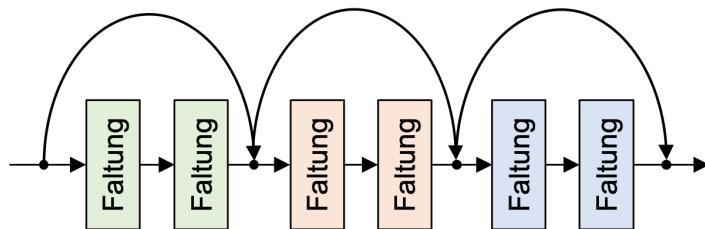


Abbildung 3.7.: Das ResNet-Konzept:

Für ein ResNet werden hauptsächlich Faltungsschichten inklusive Aktivierungsfunktion und Batch-Normalisierung verwendet. Diese Faltungsschichten werden auf klassische lineare Weise miteinander verbunden. Zusätzlich können Daten zwei Faltungsschichten mit einer Shortcut-Verbindung überspringen.

Klassifizierungs-KNN-Konzepte werden meist manuell und zufällig skaliert. Dadurch ist es schwierig, die Topologie eines KNN zu optimieren. Um eine gute Ausgangslage für die Skalierung zu schaffen, wurden mehrere unterschiedlich große Ausgangsmodelle vom EfficientNet entwickelt (EfficientNet-B0 bis EfficientNet-B7). Für die vorliegende Studienarbeit wird eine Implementierung des EfficientNet-Konzepts mit EfficientNet-B4 als Ausgangsmodell verwendet.

### Konkretisierung der Konzepte für Klassifizierungs-KNNs

Die beschriebenen Konzepte beziehen sich auf die Topologie von Klassifizierungs-KNNs und dabei besonders auf die Schichten für das Merkmalslernen. Alle oben genannten Konzepte schließen an die Schichten für das Merkmalslernen eine vollverknüpften Schicht mit 1000 Neuronen für die Klassifizierung an. Um die beschriebenen Konzepte auf die verwendeten Beispieldaten anwenden zu können, muss die letzte Schicht der Klassifizierungs-KNNs entweder ein Neuron oder zwei Neuronen besitzen.<sup>1</sup>

Aus diesem Grund wird die Topologie der oben beschriebenen Konzepte um zwei vollverknüpfte Schichten inklusive Batch-Normalisierung und ReLU-Aktivierungsfunktion erweitert. Dadurch wird die Datenmenge von 1000 auf 500 und letztlich auf zwei bzw. einen Datenpunkt reduziert. Zusätzlich wird eine „Dropout-Schicht“ verwendet, mit der 20% der in der letzten vollverknüpfte Schicht befindlichen Neuronen deaktiviert werden.

<sup>1</sup>Für eine Loss-Funktion ist ausschlaggebend, aus wie vielen Komponenten ein Ergebnis besteht (siehe Abschnitt zur Loss-Funktion).

Dadurch soll Overfitting entgegen gewirkt werden [5, 24]. Abbildung 3.8 zeigt den Aufbau der Schichten für die Klassifizierung.

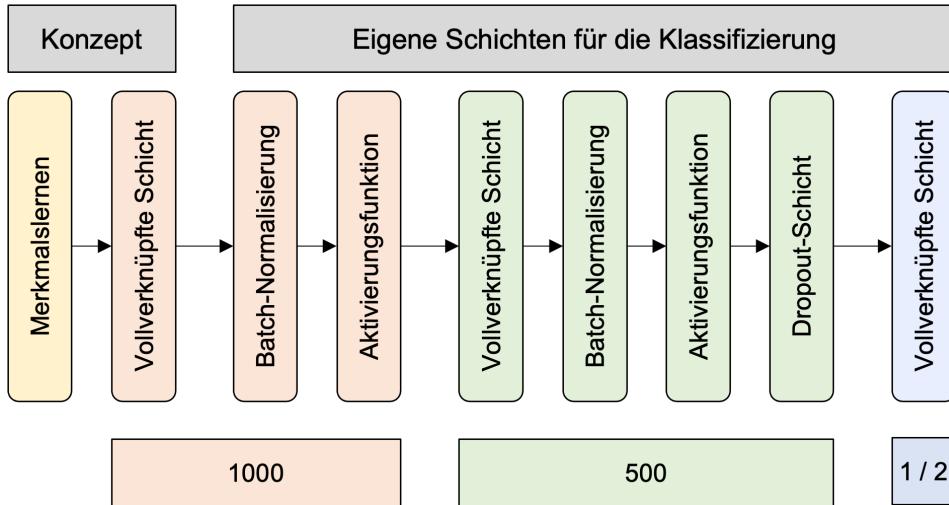


Abbildung 3.8.: Die eigenen Schichten für die Klassifizierung:

Die Topologie der Klassifizierungs-KNN-Konzepte enden alle mit einer vollverknüpften Schicht mit 1000 Neuronen. Auf diese vorgegebene Schicht folgen zwei eigene vollverknüpfte Schichten für die Klassifizierung inklusive Batch-Normalisierung und Aktivierungsfunktion. Dadurch werden die 1000 Ausgabewerte zuerst auf 500 und abschließend auf zwei bzw. einen Ausgabewert reduziert. Zusätzlich wird eine „Dropout-Schicht“ verwendet, mit der 20% der in die letzte vollverknüpften Schicht eingehenden Daten entfernt werden.

Neben der Topologie müssen für ein KNN Hyperparameter ausgewählt werden. Die Auswahl der Hyperparameter hängt stark von der gewählten Problemstellung und teilweise auch von der gewählten Topologie ab. Um die beschriebenen Konzepte an die in der vorliegenden Studienarbeit betrachtete Problemstellung anzupassen, werden nachfolgend die Hyperparameter Loss-Funktion, Trainings-Verfahren, Lernrate, Batch-Größe und Gewichts-Initialisierung festgelegt. Für jeden genannten Hyperparameter werden mindestens 2 Wahloptionen vorgestellt.

Um zu entscheiden, welche der Wahloptionen weiterführend verwendet werden soll, werden beispielhaft Klassifizierungs-KNNs aller vorgestellten Konzepte auf den echten Datensätzen für 15 Epochen trainiert und anschließend getestet. Dabei werden alle Kombinationen der Hyperparameterbelegung betrachtet. Die Tests zeigen für alle Wahloptionen die durchschnittliche Vorhersage-Genaugkeit und den mittleren Loss-Wert bei den

Vorhersagen auf den Validierungsdaten. Die Ergebnisse werden über den betreffenden Hyperparameter und die Klassifizierungs-KNN-Konzepte aggregiert und in Abhängigkeit von der Anzahl trainierter Epochen dargestellt.

### **Loss-Funktion**

Die Loss-Funktion eines KNN bestimmt die Abweichung einer Vorhersage von dem erwarteten Ergebnis für eine Datensatz. Mit dem errechneten Loss-Wert werden die Gewichte eines KNN angepasst. Die Forschung im Bereich der KNNs hat gezeigt, dass Loss-Funktionen i. d. R. für bestimmte Problemfelder geeignet sind. So ist z. B. der MSE-Loss für Regressions-Probleme und die Kreuzentropie für Klassifizierungsprobleme geeignet [5, 27].

Bei der Auswahl einer Loss-Funktion muss beachtet werden, dass die Loss-Funktion mit der Ausgabeschicht eines KNN kompatibel ist. Denn für eine Loss-Funktion ist relevant, in welcher Form die von den Schichten eines KNN errechneten Ergebnisse vorliegen. Es ist z. B. ausschlaggebend, aus wie vielen Komponenten ein Ergebnis besteht und aus welchem Intervall diese Ergebniskomponenten stammen [5, 27].

Der anvisierte Problembereich ist die Klassifizierung. In diesem Problembereich wird meistens die Kreuzentropie als Loss-Funktion eingesetzt. Die Kreuzentropie kann Loss-Werte für Ergebnisse mit einer oder mehr Komponenten berechnen. Dabei müssen die Ergebniskomponenten Wahrscheinlichkeits-Werte sein und entsprechend aus dem Intervall  $[0, 1]$  stammen. Um für eine beliebige positive Anzahl Ergebniskomponenten eine Wahrscheinlichkeitsverteilung zu errechnen, wird i. d. R. die SoftMax-Aktivierungsfunktion auf die Ausgaben der letzten Schicht eines KNN angewendet [5, 30].

Es wird ein Klassifizierungsproblem mit zwei gegensätzlichen Problemklassen (krank und gesund) betrachtet. Der Ergebnisvektor kann in diesem Fall mit einer oder zwei Komponenten dargestellt werden. Wenn nur eine Komponente verwendet werden soll, wird anstelle der klassischen Kreuzentropie die binäre Kreuzentropie als Loss-Funktion verwendet. Zusätzlich wird anstelle der Softmax-Aktivierungsfunktion die Sigmoid-Aktivierungsfunktion benutzt, um die notwendigen Wahrscheinlichkeits-Werte zu errechnen [27, 30].

Für die Klassifizierungs-KNNs der vorliegenden Studienarbeit stehen die folgenden beiden Optionen für die Auswahl einer Loss-Funktionen zur Verfügung:

- Die klassische Kreuzentropie mit der SoftMax-Aktivierungsfunktion und

- die Binäre Kreuzentropie mit der Sigmoid-Aktivierungsfunktion.

Um eine Auswahl treffen zu können, wurden Klassifizierungs-KNNs mit verschiedenen Hyperparameter-Konfigurationen auf den echten Daten trainiert und getestet. Abbildung A.1 im Anhang zeigt die Ergebnisse dieser Tests. Die Gegenüberstellung der Testergebnisse zeigt, dass die Auswahl der Loss-Funktion die Vorhersage-Genauigkeit im Durchschnitt minimal beeinflusst. Es zeigt sich, dass für alle Klassifizierungs-KNN-Konzepte die binäre Kreuzentropie etwas bessere Ergebnisse liefert, als die allgemeine Kreuzentropie. Zudem fällt auf, dass das EfficientNet-Konzept bei dieser Betrachtung die besten Vorhersage-Genauigkeiten liefert.

Der durchschnittliche Loss-Wert bei den Vorhersagen unterscheidet sich je nach Wahloption und Klassifizierungs-KNN-Konzept. Es zeigt sich, dass das EfficientNet- und das IncNet-Konzept von der Wahl der binären Kreuzentropie als Loss-Funktion profitieren. Das ResNet-Konzept weist stärkere Schwankungen über die trainierten Epochen hinweg auf, resultiert jedoch auch mit einem geringeren durchschnittlichen Loss-Wert.

Zusätzlich zu den betrachteten allgemeinen, aggregierten Testergebnissen, können speziell die Klassifizierungs-KNNs mit den besten Testergebnissen betrachtet werden. Dabei zeigt sich, dass für alle hier betrachteten Klassifizierungs-KNN-Konzepte die binäre Kreuzentropie zu den besten Testergebnissen führt. Aufgrund der beschriebenen Leistungsunterschiede wird für die Klassifizierungs-KNNs in der vorliegenden Studienarbeit die binäre Kreuzentropie als Loss-Funktion verwendet.

### Trainings-Verfahren

Das Training von KNNs wird Anfang 2022 mit Varianten des Gradientenabstiegs-Verfahrens durchgeführt. Diese Varianten sind optimierte Versionen des ursprünglichen Verfahrens und führen bei richtiger Konfiguration schneller zu besseren Trainingsergebnissen. Das Trainings-Verfahren ist nicht in der Form an die Problemstellung gebunden wie die Loss-Funktion. In der Regel können die Trainings-Verfahren auf alle Problemstellungen erfolgreich angewendet werden [5].

Das RMSprop-Verfahren ist ein Trainings-Verfahren mit dynamischen, individuellen Lernraten für jedes Neuron. Die grundlegende Variante des RMSprop-Verfahrens integriert kein Momentum. Allerdings kann dies zum RMSprop-Verfahren ergänzt werden. Auch in

seiner grundlegenden Variante ohne Momentum ist das RMSprop-Verfahren sehr effizient und effektiv [5, 30].

Das ADAM-Verfahren basiert auf dem RMSprop-Verfahren und optimiert den Trainingsprozess noch weiter. Unter anderem ist in dem ADAM-Verfahren immer ein Momentum integriert. Das ADAM-Verfahren ist Anfang 2022 eines der beliebtesten Trainings-Verfahren aufgrund seiner Effizienz, Effektivität und Robustheit [5, 30].

Für die Klassifizierungs-KNNs der vorliegenden Studienarbeit stehen die folgenden beiden Optionen zur Auswahl eines Trainings-Verfahrens zur Verfügung:

- Das ADAM-Verfahren und
- das RMSprop-Verfahren ohne Momentum.

Um eine Auswahl treffen zu können, wurden Klassifizierungs-KNNs mit verschiedenen Hyperparameter-Konfigurationen mit den echten Daten trainiert und getestet. Abbildung A.2 im Anhang zeigt die Ergebnisse dieser Tests. Die Gegenüberstellung der Testergebnisse zeigt, dass die Auswahl des Trainings-Verfahrens die Vorhersage-Genauigkeit eines Klassifizierungs-KNN im Durchschnitt nur marginal beeinflusst. Bei dieser Betrachtung führen das EfficientNet- und das IncNet-Konzept zu den besten Vorhersage-Genauigkeiten. Die betrachteten Effekte der Auswahl des Trainings-Verfahrens auf die Vorhersage-Genauigkeit finden sich bei der Betrachtung des durchschnittlichen Loss-Werts wieder.

Zusätzlich zu den betrachteten allgemeinen, aggregierten Ergebnissen, können speziell die Klassifizierungs-KNNs mit den besten Ergebnissen betrachtet werden. Dabei zeigt sich, dass für alle hier betrachteten Klassifizierungs-KNN-Konzepte das ADAM-Verfahren zu den besten Ergebnissen führt. Aufgrund der beschriebenen Leistungsunterschiede wird für die Klassifizierungs-KNNs in der vorliegenden Studienarbeit das ADAM-Trainings-Verfahren verwendet.

### Lernrate

Die Lernrate ist ein Proportionalitätsfaktor in der Lernregel des Trainings-Verfahrens, über den die Schrittweite beim Gradientenabstieg geregelt wird. Meist wird die Lernrate dynamisch während des Trainingsvorgangs angepasst, um Probleme wie das Oszillieren

um ein Minimum zu verhindern. Abhängig vom Trainings-Verfahren können Lernraten individuell für Neuronen oder universell für das ganze KNN definiert werden [5, 24].

Beide oben beschriebenen Trainings-Verfahren benutzen dynamische, individuelle Lernraten für jedes Neuron eines KNN. In einem solchen Fall muss ein Startwert für die Lernraten der Neuronen festgelegt werden, da die Anpassung der Lernrate iterativ abläuft. Der Ausgangspunkt der Lernraten wird global festgelegt und entscheidet über den Trainingserfolg und die Trainingseffizienz. Die Lernrate kann sich im Intervall von  $[0, 1]$  befinden und sollte abhängig vom betrachteten Problem gewählt werden [5, 24].

Für die Klassifizierungs-KNNs der vorliegenden Studienarbeit werden die folgenden Lernraten als Startwerte in Betracht gezogen:

- $2 \cdot 10^{-3}$ ,
- $2 \cdot 10^{-4}$  und
- $2 \cdot 10^{-5}$ .

Um eine Auswahl treffen zu können, wurden Klassifizierungs-KNNs mit verschiedenen Hyperparameter-Konfigurationen auf den echten Daten trainiert und getestet. Abbildung A.3 im Anhang zeigt die Ergebnisse dieser Tests. Die Gegenüberstellung der Testergebnisse zeigt, dass die Auswahl der Lernrate einen Einfluss auf die Vorhersage-Genauigkeit eines Klassifizierungs-KNN hat. Es ist zu erkennen, dass eine Lernrate von  $2 \cdot 10^{-3}$  zu wesentlich schlechteren Vorhersage-Genauigkeiten führt. Eine Lernrate von  $2 \cdot 10^{-5}$  führt bei dieser Betrachtung für das IncNet- und das ResNet- Konzept zu den höchsten Vorhersage-Genauigkeiten. Das EfficientNet-Konzept erzielt mit einer Lernrate von  $2 \cdot 10^{-4}$  im Durchschnitt die höchsten Vorhersage-Genauigkeiten. Diese Erkenntnisse werden durch die Betrachtung des durchschnittlichen Loss-Werts bestätigt. Auch bei dieser Betrachtung führt das EfficientNet-Konzept zu den besten Testergebnissen.

Zusätzlich zu den betrachteten allgemeinen, aggregierten Ergebnissen, können speziell die Klassifizierungs-KNNs mit den besten Ergebnissen betrachtet werden. Dabei zeigt sich, dass für das ResNet- und das IncNet-Konzept eine Lernrate von  $2 \cdot 10^{-5}$  die besten Ergebnisse liefern. Das EfficientNet-Konzept funktioniert am besten mit einer Lernrate von  $2 \cdot 10^{-4}$ . Aufgrund der beschriebenen Leistungsunterschiede wird für die ResNets

sowie die IncNets in der vorliegenden Studienarbeit eine Lernrate von  $2 \cdot 10^{-5}$  und für die EfficientNets eine Lernrate von  $2 \cdot 10^{-4}$  verwendet.

### Batch-Größe

Bei modernen Trainings-Verfahren für KNNs werden Anpassungen der Gewichte eines KNN nach der Verarbeitung einer „Batch“ genannten Teilmenge der Trainingsdatensätze vorgenommen. Dadurch wird Schritte in die falsche Richtung (aus einem Tal heraus) vorgebeugt, die durch Ausreißer-Datensätze hervorgerufen werden. Außerdem steigert dieses Vorgehen bei der Auswahl der richtigen Batch-Größe die Trainingseffizienz [5, 30].

Ist die Größe der Batches zu klein, können weiterhin Trainingsschritte in die falsche Richtung auftreten und der Rechenaufwand steigt durch häufige Gewichtsanpassungen. Zu große Batches können bewirken, dass die Gewichte nur geringfügig angepasst werden. Dadurch kann es sehr lange dauern bis das Training erfolgreich ist. Die Batch-Größe wird durch den verfügbaren VRAM der verwendeten Grafikkarten begrenzt [5, 30]. Aus diesem Grund werden in der vorliegenden Studienarbeit keine Batch-Größen über 30 betrachtet.

Für die Klassifizierungs-KNNs der vorliegenden Studienarbeit werden die folgenden Batch-Größen in Betracht gezogen:

- 20,
- 25 und
- 30.

Um eine Auswahl treffen zu können, wurden Klassifizierungs-KNNs mit verschiedenen Hyperparameter-Konfigurationen auf den echten Daten trainiert und getestet. Abbildung A.4 im Anhang zeigt die Ergebnisse dieser Tests. Die Gegenüberstellung der Testergebnisse zeigt, dass die Auswahl der Batch-Größe die Vorhersage-Genauigkeit im Durchschnitt minimal beeinflusst. Auch der durchschnittliche Loss-Wert wird im Durchschnitt nur unwesentlich durch die Auswahl der Batch-Größe beeinflusst. Das EfficientNet- und das IncNet-Konzept erzielen bei dieser Betrachtung die geringsten Loss-Werte.

Zusätzlich zu den aggregierten Ergebnissen können speziell die Klassifizierungs-KNNs mit den besten Ergebnissen betrachtet werden. Das ResNet- und das EfficientNet-Konzept erzielen mit einer Batch-Größe von 25 die besten Ergebnisse. Das IncNet-Konzept arbeitet am besten mit einer Batch-Größe von 30. Aufgrund der beschriebenen Leistungsun-

terschiede wird für die ResNets und EfficientNets eine Batch-Größe von 25 und für die IncNets eine Batch-Größe von 30 verwendet.

### Gewichts-Initialisierung

Die Gewichte von KNNs werden klassisch über einer Wahrscheinlichkeitsverteilung zufällig initialisiert. Diese Variante ist nicht an eine Problemstellung oder eine KNN-Architektur gebunden. Allerdings wurde dieser Ansatz häufig durch die Auswahl anderer Wahrscheinlichkeitsverteilungen an eine Architektur angepasst [27].

„Pretraining“ ist eine andere Methode zur Initialisierung der Gewichte eines KNN. Beim Pretraining werden die Gewichte eines KNN so initialisiert, dass sie für eine bestimmte Problemstellung möglichst wenig angepasst werden müssen. Dazu werden die Gewichte eines KNN mit den Daten der betreffenden Problemstellung vorgenommen. Dadurch werden die Gewichte eines KNN für die Erkennung der Merkmale dieser Problemstellung angepasst [5, 54].

Ein solches Pretraining kann auch für „Transfer-Lernen“ verwendet werden. Dabei werden die vorgenommenen KNNs für die Bearbeitung einer anderen ähnlichen Problemstellung verwendet. Ein solches Transfer-Lernen kann besonders beim Merkmals-Lernen von CNNs von Vorteil sein. Denn die Merkmale geringer Komplexität wie z. B. Linien in Bildern können vielfältig verwendet werden [25, 54]. In der vorliegenden Studienarbeit werden Klassifizierungs-KNN-Konzepte betrachtet, die für die ILSVCR entwickelt wurden. Die für die vorliegende Studienarbeit verwendeten Implementierungen dieser Konzepte von Pytorch können mit zufällig initialisierten oder auf ImageNet Datensätzen vorgenommenen Gewichten verwendet werden. In der vorliegenden Studienarbeit wird nur das Pretraining in Form von Transfer-Lernen betrachtet.

Für die Klassifizierungs-KNNs der vorliegenden Studienarbeit werden die folgenden Varianten zur Initialisierung von Gewichten in Betracht gezogen:

- Zufällige Initialisierung mit Werten einer Wahrscheinlichkeitsverteilung und
- Initialisierung mit Werten aus Pretraining (Transfer-Lernen).

Um eine Auswahl treffen zu können, wurden Klassifizierungs-KNNs mit verschiedenen Hyperparameter-Konfigurationen auf den echten Daten trainiert und getestet. Abbildung A.5 im Anhang zeigt die Ergebnisse dieser Tests. Die Gegenüberstellung der Testergeb-

nisse zeigt, dass durch Pretraining im Durchschnitt bessere Vorhersage-Genauigkeiten und Loss-Werte erzielt werden als bei einer zufälligen Gewichts-Initialisierung. Die EfficientNets erzielen auch hier die besten Ergebnisse.

Zusätzlich zu den betrachteten allgemeinen, aggregierten Ergebnissen, können speziell die Klassifizierungs-KNNs mit den besten Ergebnissen betrachtet werden. Dabei zeigt sich, dass für alle hier betrachteten Klassifizierungs-KNN-Konzepte ein Pretraining zu den besten Ergebnissen führt. Aufgrund der beschriebenen Leistungsunterschiede werden die Gewichte der Klassifizierungs-KNNs in der vorliegenden Studienarbeit durch Pretraining initialisiert.

### 3.3.1. Die Klassifizierungs-KNN-Modelle

Es wurden die Topologien von drei verschiedenen Klassifizierungs-KNN-Konzepten beschrieben und durch eigene Schichten für die Klassifizierung an die Problemstellung der vorliegenden Studienarbeit angepasst. Anschließend wurden Hyperparameter für alle drei Klassifizierungs-KNN-Konzepte festgelegt. Die drei entstandenen Klassifizierungs-KNN-Modelle werden in Abbildung 3.9 noch einmal kompakt gegenübergestellt.

Eigenschaft	ResNet	IncNet	EfficientNet
<b>Topologie</b>			
Merkmalslernen	ResNet18	IncNet V3	EfficientNet-B4
Klassifizierung	eigene Schichten für die Klassifizierung		
<b>Hyperparameter</b>			
Loss-Funktion	Binäre Kreuzentropie		
Trainings-Verfahren	ADAM-Verfahren		
Lernrate	$2 * 10^{-5}$	$2 * 10^{-5}$	$2 * 10^{-4}$
Batch-Größe	25	30	25
Gewichts-Initialisierung	Pretraining		

Abbildung 3.9.: Eine Aufstellung der drei Klassifizierungs-KNN-Modelle

# 4. Umsetzung und Implementierung

In dem vorliegenden Kapitel 4 wird beschrieben, wie und mit welchen Hilfsmitteln die in Kapitel 3 vorgestellten Konzepte umgesetzt werden. Es wird zuerst in Unterkapitel 4.1 auf die allgemeinen technischen Hilfsmittel eingegangen, die für die Implementierung der Konzepte verwendet werden. Dabei geht es um die Software-Werkzeuge, mit denen die Programme verfasst und ausgeführt werden können. Außerdem wird die zentrale Software-Bibliothek betrachtet, mit der die Machine Learning Komponenten entwickelt werden. Unterkapitel 4.2 behandelt den Quellcode, mit dem die Laufzeitumgebungen von Google Colaboratory (Colab) auf das Training von GANs und Klassifizierungs-KNNs vorbereitet werden. Anschließend wird in Unterkapitel 4.3 die Implementierung der in Unterkapitel 3.2 vorgestellten Konzepte für GANs beschrieben. Unterkapitel 4.4 behandelt die Implementierung der Konzepte für Klassifizierungs-KNNs aus Unterkapitel 3.3.

## 4.1. Technische Hilfsmittel

### 4.1.1. Software-Werkzeuge

Die Implementierung der GANs und Klassifizierungs-KNNs werden in der Programmiersprache Python vorgenommen. Denn diese Programmiersprache ist im Machine Learning Umfeld etabliert. Es gibt für Python viele Software-Bibliotheken, mit denen Machine Learning betrieben wird. Zudem sind große Bereiche im Machine Learning für Python im Internet dokumentiert [55].

Im Machine Learning Umfeld wird die Programmiersprache Python oft mit Jupyter Notebooks in Verbindung gebracht. Jupyter Notebooks bestehen aus Zellen, die Platz für Code in unterschiedlichen Programmiersprachen, Markdown-Text oder Medieninhalte bieten. Mehrere Zellen können in Abschnitte gruppiert und dadurch inhaltlich abgegrenzt werden. Der Quellcode in Zellen und auch ganzen Abschnitten kann separat ausgeführt werden [56, 57].

Jupyter Notebook ist eine kostenlose und frei verfügbare Technologie, die von vielen Unternehmen für eigene Produkte verwendet wird [57]. Ein solches Produkt ist die Web-Anwendung Google Colab, mit der Jupyter Notebooks über den Browser bearbeitet und auf Servern von Google betrieben werden können. Dabei variiert die von den Servern zur Verfügung gestellte Rechenleistung und Speicherkapazität. Colab stellt verschiedene Laufzeitumgebungs-Typen zur Verfügung. Unter anderem ist es möglich, hoch performante Grafikkarten zu verwenden, die das Training von KNNs ungemein beschleunigen. Colab kann kostenlos verwendet werden. Durch Abschluss eines kostenpflichtigen Abonnements wird dem Benutzer Zugang zu leistungsstärkerer Hardware und größerer Speicherkapazität gewährt.

Auf den Server-Laufzeitumgebungen von Colab sind zahlreiche Software-Bibliotheken installiert. Besonders Machine Learning Software-Bibliotheken wie z. B. Tensorflow, Pytorch, NumPy, Pandas und Matplotlib sind bereits vorhanden. Zudem werden die installierten Software-Bibliotheken automatisch aktualisiert. Sollten es dennoch vorkommen, dass eine Software-Bibliothek fehlt, kann diese nachinstalliert werden.

Colab ist in das Google-Ökosystem eingebettet. Die mit Colab erstellten und bearbeiteten Jupyter Notebooks werden in dem Google Drive gespeichert und können auch auf den Google Drive zugreifen. Dort können Jupyter Notebooks wie jede andere Datei verwaltet werden. Das beinhaltet auch das Teilen mit anderen Personen. Dadurch können mehrere Benutzer parallel an einem Jupyter Notebook arbeiten. Die Ausführung von Quellcode ist allerdings nicht parallel möglich, da ein Jupyter Notebook an eine Laufzeitumgebung gebunden ist.

Aufgrund der beschriebenen Eigenschaften von Jupyter Notebooks und Colab, wird in der vorliegenden Studienarbeit für die Implementierung der GANs und Klassifizierungs-KNNs Colab verwendet.

#### 4.1.2. Machine Learning Software-Bibliothek

Für die Implementierung der GANs und der Klassifizierungs-KNNs stehen die zwei bekannten Machine Learning Frameworks „Tensorflow“ und „PyTorch“ zur Auswahl. Das Open Source Machine Learning Framework PyTorch wird vom Facebook-Forschungsteam für Künstliche Intelligenz (KI) entwickelt und ist in der Forschung das aktuell beliebteste

Framework für Machine Learning Modelle [58, 59]. Tensorflow ist ein von Google entwickeltes Konkurrenz Framework für Machine Learning. Im Hinblick auf die Forschung bietet PyTorch gegenüber TensorFlow verschiedene Vorteile, weshalb sich PyTorch in der Forschung durchsetzen konnte. PyTorch ist einfacher zu benutzen und besitzt eine verständlichere API als Tensorflow. Tensorflow hat im Laufe der Zeit immer wieder neue APIs eingeführt, wodurch der Einstieg in das Framework erschwert wurde. In der Forschung und der vorliegenden Studienarbeit steht die Entwicklung neuer Ideen und Konzepte im Fokus. Daher wird ein einfacher Einstieg in das Framework bevorzugt.

Der andere entscheidende Unterschied zwischen den zwei Frameworks ist die Art und Weise wie die Modelle ausgeführt werden. Bei beiden Frameworks wird das Machine Learning Modell in Form eines Graphen erstellt. Jedoch unterscheiden sich die Frameworks im Hinblick darauf, wie die Graphen erstellt werden. PyTorch basiert auf dem „define-by-run“ Prinzip, bei dem die Graph-Struktur des Modells zusammen mit der Berechnung definiert wird [59]. Da der Graph während der Ausführung definiert wird, handelt es sich um einen dynamischen Graphen. Das Vorgehen entspricht der natürlichen Softwareentwicklung.

TensorFlow hingegen basiert auf einem Konzept mit statischen Graphen [59]. Im ersten Schritt wird der gesamte Graph definiert und gespeichert. Im zweiten Schritt werden Daten an den Graphen übergeben und die definierten Berechnungen werden durchgeführt. Dieses Vorgehen entspricht nicht der natürlichen Softwareentwicklung und bringt vor allem beim Debuggen Probleme mit sich [59]. Durch den zuerst berechneten Graphen geht der Zusammenhang zwischen einem Laufzeitfehler und der auslösenden Stelle im Code verloren. Ein schrittweises Debuggen des Codes ist nicht mehr möglich, da nur der Graph ausgeführt wird.

Durch diese Unterschiede in der Ausführung, ist PyTorch intuitiver und einfacher zu nutzen und konnte sich in der Forschung besser etablieren [59]. Mit der Variante des statischen Graphen bietet Tensorflow aber auch Vorteile, die vor allem in der Industrie wichtig sind. Da zuerst der gesamte Graph definiert wird, kann dieser im Anschluss optimiert werden. Dadurch kann die Performanz von KNNs verbessert und Kosten eingespart werden [59]. Des Weiteren ist für die Ausführung des Graphen keine Python-Umgebung mehr nötig. Das erleichtert den Einsatz auf mobilen Endgeräten und Servern [59]. Tabelle 4.1 stellt beide Frameworks vergleichend nebeneinander.

PyTorch	Tensorflow
<ul style="list-style-type: none"> <li>• PyTorch ist das in der Forschung meist verwendete Framework</li> <li>• Einfach zu benutzen aufgrund einer gut verständlichen API</li> <li>• PyTorch arbeitet nach dem „define-by-run“ Prinzip und erzeugt dynamische Graphen</li> <li>• Einfaches Debugging aufgrund der dynamischen Graphen</li> <li>• Der Graph kann nur mithilfe von TorchScript optimiert und auf Servern oder mobilen Endgeräten ausgeführt werden</li> </ul>	<ul style="list-style-type: none"> <li>• Tensorflow wird hauptsächlich in der Industrie genutzt</li> <li>• Es existieren unterschiedliche APIs für die Nutzung des Frameworks</li> <li>• Tensorflow verwendet statische Graphen (dynamische Graphen sind mit dem Eager-Modus in Tensorflow 2.0 verfügbar)</li> <li>• Debugging ist umständlich aufgrund der statischen Graphen</li> <li>• Der statische Graph kann optimiert werden und eignet sich gut für die Ausführung auf Servern und mobilen Endgeräten</li> </ul>

Tabelle 4.1.: Vor- und Nachteile der Machine Learning Frameworks PyTorch und Tensorflow im Überblick.

Anfang 2022 sind die Schwachstellen der zwei Frameworks teilweise behoben. So hat Google mit TensorFlow 2.0 einen sogenannten Eager-Modus eingeführt, der ebenfalls nach dem „define-by-run“ Prinzip funktioniert. Facebook hat mit TorchScript eine Zwischenrepräsentation eingeführt, die im Prinzip ein starker Graph ist. Damit haben beide Unternehmen den Schwachstellen ihrer Frameworks entgegen gewirkt. Dennoch ist PyTorch nach wie vor intuitiver zu nutzen und wird in der Forschung bevorzugt. Aufgrund der genannten Vorteile wurde entschieden, PyTorch für die Implementierungen der vorliegenden Studienarbeit zu verwenden.

## 4.2. Vorbereiten der Laufzeitumgebung

In diesem Unterkapitel werden die Bestandteile eines Jupyter Notebooks beschrieben, mit denen die Laufzeitumgebungen von Google Colab auf die Ausführung weiteren Quell-

Codes vorbereitet werden. Das umfasst das Herunterladen und Vorbereiten der Daten, Importieren von Software-Bibliotheken und die Spezifizierung des Laufzeitumgebungs-Typs. Diese Aufgaben müssen sowohl für die Implementierung der GANs als auch für die Klassifizierungs-KNNs erledigt werden. Dementsprechend befinden sich die nachfolgend beschriebenen Zellen am Anfang aller Jupyter Notebooks der vorliegenden Studienarbeit.

### **Importieren von Software-Bibliotheken**

Die Laufzeitumgebungen der Web-Anwendung Google Colab enthalten eine große Menge von Python-Paketen. Um auf die Funktionen dieser Pakete zugreifen zu können, müssen die Pakete importiert werden. Für die vorliegende Studienarbeit müssen viele Bestandteile von Pytorch importiert werden. Zudem werden „NumPy“ für die Matrizenrechnung und „Matplotlib“ für die Darstellung von Bildern benötigt. Abschließend werden Software-Bibliotheken für allgemeine System-Funktionen wie das Verarbeiten von Dateien unterschiedlicher Formate benötigt.

### **Datensatz herunterladen**

Die echten Daten für das Training und das Testen der Klassifizierungs-KNNs werden von der Web-Anwendung „Kaggle.com“ bereitgestellt. Mit dem Kommandozeilenwerkzeug „kaggle“<sup>1</sup> können die Daten automatisiert von einem Jupyter Notebook in die Laufzeitumgebung heruntergeladen werden.

Damit Kaggle.com den Zugriff auf seine Datenbestände gewährt, muss ein „API Token“ angegeben werden. Dieser API Token kann in Form der Datei „Kaggle.json“ auf der Website von Kaggle.com erzeugt werden. Damit die Laufzeitumgebung auf diesen API-Token zugreifen kann, muss er in die Laufzeitumgebung kopiert werden. Um diesen Kopiervorgang zu automatisieren, wird ein Google Drive Verzeichnis mit der Kaggle.json-Datei in die Laufzeitumgebung gemounted. Von dort wird die Datei automatisiert mit einfachen Kommandozeilenbefehlen in die Laufzeitumgebung kopiert.

---

<sup>1</sup>Das Kommandozeilenwerkzeug „kaggle“ ist nicht von Grund auf in den Laufzeitumgebungen von Google Colab enthalten. Daher muss das Kommandozeilenwerkzeug zuerst mit dem Paket-Manager „pip“ installiert werden.

Die in der vorliegenden Studienarbeit verwendeten Daten werden von Kaggle.com in Form eines Zip-Archivs bereitgestellt. Um die Daten zu verwenden, muss das Zip-Archiv entpackt werden. Dazu wird das Kommandozeilenwerkzeug unzip verwendet.

### **Pytorch für die Nutzung von Grafikkarten konfigurieren**

Für die vorliegende Studienarbeit wird die Rechenleistung von Grafikkarten verwendet. Damit das hier beschriebenen Jupyter Notebook seine Berechnungen auf einer Grafikkarte ausführt, müssen die relevanten Objekte wie z. B. KNN-Modelle, Bild-Tensoren, Label-Tensoren in den VRAM der Grafikkarte geladen werden.

Damit alle Tensoren automatisch in den VRAM einer Grafikkarte geladen werden, wird in diesem Abschnitt der Standard-Typ für Tensoren auf „`torch.cuda.FloatTensor`“ festgelegt. Das Schlüsselwort „`cuda`“ weist darauf hin, dass der Tensor im VRAM einer Grafikkarte vorgehalten werden soll.

Damit in den nachfolgenden Abschnitten eines Jupyter Notebooks ein KNN-Modell in den VRAM einer Grafikkarte geladen werden kann, wird in diesem Abschnitt die Variable „`device`“ definiert. Diese Variable repräsentiert den Typ des Geräts mit dem eine Laufzeitumgebung Berechnungen durchführt. Basierend auf dem gewählten Gerät-Typ wird ein Objekt in den VRAM einer Grafikkarte (Typ „`cuda`“) oder in den Arbeitsspeicher der Laufzeitumgebung (Typ „`cpu`“) geladen. Für die vorliegende Studienarbeit wird der Typ `cuda` verwendet.

## **4.3. Implementierung der GANs**

In diesem Kapitel werden die Quellcode-Abschnitte für die Implementierung der GANs beschrieben. Das umfasst das Bereitstellen der Datensätze, das Erstellen von Generator und Diskriminatior sowie das Training der GANs. Das Jupyter Notebook ist dafür in inhaltlich getrennte Abschnitte aufgeteilt, die nachfolgend beschrieben werden.

## Datensatz vorbereiten

Die Dataset-Klasse bereitet den Datensatz für das Training vor. Beim Erzeugen eines Dataset-Objekts werden die Bilder und die dazugehörigen Labels in den Arbeitsspeicher eingelesen. Um die gespeicherten Trainingsbeispiele einsehen zu können, besitzt die Klasse eine Methode, die die Bilder aus den Trainingsdaten visualisiert. Diese Funktion wird verwendet, um Vergleiche zwischen den Trainingsdaten und den generierten Bildern vorzunehmen.

Für das Training besitzt das Dataset-Objekt eine Methode, die ein Trainingsbeispiel zusammen mit dem zugehörigen One-Hot-codierten Label zurück gibt. Dabei wird das Bild auf 300x300 Pixel zugeschnitten und in einen Tensor konvertiert. Anschließend werden die Pixelwerte des Bilds auf das Intervall [-1,1] skaliert.

## Hilfsfunktionen

In diesem Abschnitt wird eine Klasse und eine Hilfsmethode für die Verarbeitung der Daten und das Training eines GAN erstellt. Die wichtigste Funktion hat die Klasse View. Die Klasse View ist ein selbst erstelltes PyTorch Modul und transformiert eine Eingabe in eine neue Dimension, die bei der Initialisierung festgelegt werden kann. Da die Klasse View ein PyTorch Modul ist, können View-Objekte als Schicht in ein KNN integriert werden. Mit View-Objekten können z. B. Merkmalskarten von Faltungsschichten für die Verarbeitung mit vollverknüpften Schichten transformiert werden.

Die in diesem Abschnitt definierte Hilfsmethode liefert einen Tensor aus zufälligen, normalverteilten Werten zurück. Die Größe des Tensor kann als Parameter an die Methode übergeben werden. Diese Methode wird für das Generieren der zufälligen Startwerte des Generators verwendet.

## Generator

Die Implementierung des Generators folgt dem in Unterkapitel 3.2 vorgestellten Konzept für einen Generator. Zusätzlich wird eine Methode implementiert, die die Trainingsschritte für den Generator umsetzt. In der Trainingsmethode werden zuerst die

zufälligen Startwerte zusammen mit einem zufälligen Label an das Generator-Netzwerk übergeben. Anschließend wird das generierte Bild an den Diskriminatator übergeben. Mit Hilfe der Fehlerfunktion wird dann der Fehler aus dem zufälligen Label und der Ausgabe des Diskriminators berechnet. Anschließend werden die Gewichte angepasst. In regelmäßigen Abständen wird der berechnete Loss-Wert abgespeichert, um den Verlauf der Loss-Werte zu dokumentieren. Der Verlauf des Loss-Werts wird benötigt, um Aussagen über den Trainingsfortschritt des Generators treffen zu können.

## Diskriminatator

Die Implementierung des Diskriminators folgt dem Konzept aus Unterkapitel 3.2. Der Diskriminatator besitzt genau wie der Generator eine Trainingsmethode, die alle Trainingsschritte umsetzt. Für das Training des Diskriminators wird zuerst das echte Bild zusammen mit dem zugehörigen Label an das Netz übergeben. Die Ausgabe des Diskriminators wird zwischengespeichert. Anschließend werden zufällige Startwerte für den Generator erzeugt und zusammen mit einem zufälligen Label an den Generator übergeben, um ein Bild zu erzeugen. Dieses synthetische Bild wird zusammen mit dem zufällig erzeugten Label an den Diskriminatator übergeben. Die Ausgabe des Diskriminators wird zwischengespeichert. Im Anschluss werden die Loss-Werte für die Vorhersagen auf dem echten und dem synthetischen Bild berechnet. Die zwei Ergebnisse werden miteinander verrechnet und der kombinierte Loss-Wert wird genutzt, um die Gewichte des Diskriminators zu aktualisieren. Der Verlauf der berechneten Loss-Werte wird genau wie beim Generator dokumentiert, um eine Aussage über den Trainingsfortschritt des Diskriminators treffen zu können.

## Training

Für das Training der GANs werden zuerst jeweils ein Generator und ein Diskriminatator Objekt erzeugt. Beide Objekte werden anschließend auf den am Anfang des Jupyter Notebooks definierten Geräte-Typen verschoben. In der Trainingsschleife wird über die Trainingsdatensätze iteriert. Dabei werden die Datensätze an die entsprechenden Trainingsmethoden von Generator und Diskriminatator übergeben.

## 4.4. Implementierung der Klassifiziererungs-KNNs

In diesem Unterkapitel wird der Quellcode beschrieben, mit dem die Klassifiziererungs-KNNs aus Unterkapitel 3.3 implementiert und trainiert werden. Das umfasst das bereitstellen von Datensätzen, das Festlegen der Topologie und der Hyperparameter sowie das Training und Testen der drei Klassifiziererungs-KNNs. Für diese Implementierungen wurde ein Jupyter Notebook mit inhaltlich abgegrenzten Abschnitten angelegt. Diese Abschnitte werden nachfolgend beschrieben.

### 4.4.1. Allgemeine Variablen und Methoden

Dieser Abschnitt des Jupyter Notebooks beinhaltet Quellcode, der inhaltlich nicht einem einzigen Abschnitt zugeordnet werden kann. Das hier beschriebene Jupyter Notebook beherbergt in diesem Abschnitt lediglich eine Variable mit Namen „real\\_syn\\_switch“. Diese Variable bestimmt, ob in dem Jupyter Notebook synthetische oder echte Daten für das Training verwendet werden. In den nachfolgenden Abschnitten werden Code-Zeilen oder sogar ganze Zellen nur für das Training mit synthetischen Daten benötigt. Die betroffenen Code-Bestandteile können durch den Wert „real“ deaktiviert werden.

### 4.4.2. GAN-Generatoren

In diesem Abschnitt des Jupyter Notebooks befindet sich der Quellcode für die GAN-Generatoren. In diesem Jupyter Notebook werden die GAN-Generatoren für das Erzeugen von synthetischen Datensätzen verwendet. Es werden keine Diskriminatoren benötigt, mit denen die Generatoren trainiert werden können. Der Abschnitt enthält lediglich die Klasse „View“ und die Implementierung der Generatoren des BCE-CGAN-Konzepts und des WCGAN-Konzepts. Dabei wurden die Klassen der GAN-Generatoren auf das Nötigste reduziert. Genauere Erläuterungen zu dem Quellcode in diesem Abschnitt befinden sich in Unterkapitel 4.3.

### 4.4.3. Datensätze

Dieser Abschnitt des Jupyter Notebooks erzeugt Datensätze aus den heruntergeladenen oder generierten Daten, die für das Training von Klassifizierungs-KNNs verwendet werden können. Nachfolgend werden die Zellen dieses Abschnitts beschrieben.

#### Hilfsmethoden und -variablen

In der ersten Zelle dieses Abschnitts werden zwei für diesen Abschnitt relevante Hilfsmethoden deklariert. Die erste Methode wird verwendet, um Bilder zu zuschneiden. Dabei bleibt der Mittelpunkt des Bilds gleich. Die zweite Methode erzeugt ein Dictionary-Objekt, mit dem einem Bild der echten Testdaten das richtige Label zuordnet wird. Das ist bei den Testdaten notwendig, da die Labels getrennt von den zugehörigen Bildern in einer csv-Datei festgehalten wurden. Bei den Trainingsdaten wird keine vergleichbare Methode benötigt, da die Labels über den Dateipfad eines Bilds ermittelt werden.<sup>2</sup>

Neben den beiden Methoden werden in dieser Zelle vier Variablen definiert. Die ersten beiden Variablen speichern die Pfade zu den echten Trainings- und Testdaten. In einer separaten Variable wird festgehalten, auf welche Größe die Bilder der Datensätze zugeschnitten werden sollen. Diese Variable wird für das ResNet- und das EfficientNet-Konzept auf 300 festgelegt. Für die Implementierung des IncNet-Konzepts von Pytorch muss die Variable auf 299 gesetzt werden. Die letzte Variable bestimmt die Anzahl der Ergebniskomponenten, die ein Label besitzen muss. Je nach verwendeter Loss-Funktion müssen die Labels eine oder zwei Komponenten haben (siehe Unterkapitel 3.3).

#### Dataset-Klasse

Die zweite Zelle dieses Abschnitts enthält die Klasse „Dataset“. Ein Dataset-Objekt enthält alle Datensätze (Bilder und Labels) für einen Anwendungsfall. Es werden separate Dataset-Objekte für die Trainings-, Validierungs- und Testdatensätze erstellt. Um ein Dataset-Objekt mit Datensätzen zu befüllen, können Bilder einzeln aus Dateien oder

---

<sup>2</sup>Die Bilder der echten Trainingsdaten liegen in unterschiedlichen Verzeichnissen. Die Bezeichnung des Verzeichnisses gibt das Label der enthaltenen Bilder an. Dabei steht die Bezeichnung „hem“ für gesunde Zellen und die Bezeichnung „all“ für an Leukämie erkrankte Zellen.

gesammelt aus einem Verzeichnis eingelesen werden. In beiden Fällen muss das Label angegeben werden, das dem Bild bzw. den Bildern zugewiesen werden soll. Die Labels werden dabei als Zahl 0 für „gesund“ oder Zahl 1 für „krank“ übergeben. Das Einlesen der Bilder umfasst immer das Zuschneiden der Bilder.

Die Dataset-Objekte werden in den hinteren Abschnitten verwendet, um Dataloader-Objekte zu befüllen. Aus diesem Grund müssen die Methoden „`__len__()`“ und „`__getitem__(index)`“ von der Dataset-Klasse implementiert werden. Mit „`__len__()`“ wird die Anzahl der Datensätze in dem Dataset-Objekt zurückgegeben. Die Methode „`__getitem__(index)`“ gibt den Datensatz an der Stelle des angegebenen Indizes zurück. Dabei werden die Bilder einer Normalisierung unterzogen, sodass sich der Wertebereich der Bildpunkte nur moderat von Bild zu Bild unterscheidet. Zudem werden die Labels auf das richtige Format gebracht.

Je nach verwendeter Loss-Funktion müssen die Labels eine oder zwei Komponenten haben (siehe Unterkapitel 3.3). Die Komponenten der Labels können in beiden Fällen entweder den Wert 0 oder den Wert 1 annehmen. Bei einem Label mit einer Komponente steht der Wert 1 für eine kranke und der Wert 0 für eine gesunde Zelle. Bei Labels mit zwei Komponenten repräsentieren die Komponenten die beiden Zustände „krank“ und „gesund“. Die Komponente mit dem Wert 1 gibt den Wert des Labels an. Die Dataset-Klasse bietet die Möglichkeit einen Datensatz zu visualisieren. Dabei werden die Bilder mit dem zugehörigen Label von der Software-Bibliothek Matplotlib angezeigt.

## SynDataset-Klasse

Die oben beschriebene Dataset-Klasse kann von sich aus nur die bestehenden echten Daten aus Dateien auslesen. Um synthetische Datensätze neu zu erzeugen, wird in der dritten Zelle dieses Abschnitts die Klasse „SynDataset“ definiert. Die Klasse Dataset vererbt ihre Methoden an die Klasse SynDataset. Daher müssen lediglich die Funktionen für das Generieren von Bildern hinzugefügt werden.

Für das Generieren synthetischer Datensätze wurden GANs verschiedener Konzepte trainiert und der Zustand des Generators abgespeichert. Dieser Zustand liegt in Dateiform für alle in Unterkapitel 3.2 geschilderten GAN-Konzepte in einem Google Drive Verzeichnis. Beim Instanziieren eines SynDataset-Objekts muss angegeben werden, welche

Generator-Klasse verwendet und welcher Zustand wiederhergestellt werden soll. Zusätzlich muss angegeben werden, wie viele synthetische Datensätze erzeugt werden sollen.

Es wird ein Generator-Objekt erzeugt und die angegebene Anzahl synthetischer Datensätze generiert. Für das Generieren von synthetischen Datensätzen mit CGANs wird ein Label und ein zufälliger Startwert benötigt. Der erste Schritt beim Erzeugen eines synthetischen Datensatzes ist die Wahl eines Labels. In der vorliegenden Studienarbeit werden jeweils 50% Bilder von gesunden (Label 0) und kranken (Label 1) Zellen erzeugt. Um das gewählte Label in den GAN-Generator einzuspeisen, muss das Label One-Hot-codiert werden.

Anschließend wird ein zufälliger Startwert generiert. Label und Startwert werden zusammen in den GAN-Generator eingespeist und zu einem Bild verarbeitet. Dieses Bild wird auf die richtige Größe zugeschnitten und zusammen mit dem Label als Datensatz dem Dataset-Objekt hinzugefügt.

### **Instanziierung der Dataset-Objekte**

Es müssen Dataset-Objekte für das Training (Train-Dataset), die Validierung (Validation-Dataset) und das Testen (Test-Dataset) von Klassifizierungs-KNNs erzeugt werden. Für die Validierung und das Testen werden in jedem Fall echte Daten verwendet. Die für das Training verwendeten Daten können allerdings auch synthetisch oder eine Mischung aus beidem sein. Um diese Fälle zu unterscheiden, wird die oben definierte Variable „real\_syn\_switch“ verwendet.

Das Train-Dataset wird vorerst mit den echten Daten befüllt. Von diesen echten Trainingsdatensätzen werden 10% für die Validierung verwendet. Dazu wird eine eigenes Validation-Dataset erzeugt und mit zufälligen Datensätzen aus dem Train-Dataset befüllt. Die gewählten Datensätze werden aus dem Train-Dataset entfernt, damit diese Datensätze dem trainierten Klassifizierungs-KNN unbekannt sind.

Wenn das Training der Klassifizierungs-KNNs mit synthetischen Datensätzen durchgeführt werden soll, wird das Train-Dataset nach der Befüllung des Validation-Datasets mit einem SynDataset-Objekt überschrieben. Für eine Mischung aus echten und synthetischen Trainingsdatensätzen, kann die Variable „real\_syn\_switch“ auf „mix“ gesetzt werden. In diesem Fall wird das bereits bestehende Train-Dataset vor dem Überschrei-

ben zwischengespeichert. In das neue SynDataset werden anschließend die zwischengespeicherten Datensätze des alten Train-Datasets eingefügt.

Die Datensätze für das Test-Dataset liegen als zusätzliche Dateien in einem separaten Verzeichnis vor. Bei den Testdaten sind die Labels von den Bildern getrennt in einer csv-Datei gespeichert. Mit der vorher definierten Hilfsmethode wird ein Dictionary-Objekt erzeugt, das einem Bild das passende Label zuweist. Mit diesem Hilfsmittel können die Testdatensätze eingelesen werden.

#### 4.4.4. Klassifizierungs-KNNs

In diesem Abschnitt des Jupyter Notebooks werden die konzipierten Klassifizierungs-KNNs implementiert. Nachfolgend werden die Zellen dieses Abschnitts beschrieben.

##### Die CNN-Klasse

Die in dieser Zelle definierte CNN-Klasse kann als abstrakte Klasse betrachtet werden, da sie die Grundlage für die spezifischen Klassifizierungs-KNN-Klassen ist, aber nicht funktionsfähig instanziert werden kann. Die CNN-Klasse legt die in Unterkapitel 3.3 beschriebenen Schichten für die Klassifizierung und die Hyperparameter mit Ausnahme der Batch-Größe fest. Die Schichten für das Merkmalslernen werden durch die CNN-Klasse nicht definiert. Allerdings wird die Methode definiert, mit der diese Schichten festgelegt werden können.

Die CNN-Klasse erweitert die Klasse „nn.Module“ von Pytorch. Die „forward()“-Methode von nn.Module führt eine Vorwärtspropagierung durch eine KNN durch. Die Methode wird in der CNN-Klasse so überschrieben, dass die definierten Schichten für das Merkmalslernen und die Klassifizierung verwendet werden.

Die CNN-Klasse definiert separate Methoden für das Training und das Testen eines Klassifizierungs-KNN. Die Trainings-Methode bearbeitet Batches von Datensätzen. Die Test-Methode bearbeitet einzelne Datensätze. Beide Methoden lassen die eingehenden Datensätze vorwärts durch das KNN propagieren und berechnen anschließend den Loss-Wert. Die Methode für das Training eines Klassifizierungs-KNN führt anschließend mit

dem errechneten Loss-Wert eine Rückwärtspropagierung durch und passt die Gewichte des Klassifizierungs-KNN an. Die Methode für das Testen eines Klassifizierungs-KNN berechnet, ob das Ergebnis der Vorwärtspropagierung dem Label entspricht.

Die CNN-Klasse definiert zusätzlich eine Methode mit der die Konfiguration der Hyperparameter als String zurückgegeben wird. Das wird in den nachfolgenden Abschnitten besonders für das Abspeichern von Klassifizierungs-KNNs verwendet.

### Die Implementierungen der KNN-Konzepte

In den drei folgenden Zellen dieses Abschnitts werden die konkreten Implementierungen der abstrakten CNN-Klasse definiert. Diese konkreten Implementierungen sind die Repräsentationen der in Unterkapitel 3.3 aufgestellten Klassifizierungs-KNNs. Alle Klassen der konkreten Klassifizierungs-KNNs erben die Funktionalitäten der Superklasse „CNN“. Jede Sohnklasse überschreibt die Methode, mit der die Schichten für das Merkmalslernen festgelegt werden. Dabei werden in allen drei Sohnklassen die Pytorch-Implementierungen des jeweiligen Konzepts verwendet. Für das IncNet-Konzept wird das Inception Neural Network V3 verwendet. Das ResNet-Konzept wird durch das ursprüngliche Residual Neural Network mit 18 Schichten für das Merkmalslernen repräsentiert. Für das EfficientNet-Konzept wird das EfficientNet-B4 eingesetzt.

Die drei Sohnklassen weichen sonst nicht von der Superklasse CNN ab. Die Ausnahme ist die Implementierung des IncNet-Konzepts. Hier tritt aufgrund der Implementierung von Pytorch in manchen Fällen das Problem auf, dass die Schichten für das Merkmalslernen keinen Tensor als Ausgabe produzieren. Die anschließenden Schichten für die Klassifizierung benötigen einen solchen Tensor als Eingabe. Um dieses Problem zu lösen, werden die Ausgaben der Schichten für das Merkmalslernen in der forward()-Methode bei Bedarf in einen Tensor umgewandelt.

#### 4.4.5. Training und Testen

Mit dem Quellcode aus diesem Abschnitt des Jupyter Notebooks wird das Training von Klassifizierungs-KNNs durchgeführt und überwacht. Nachfolgend werden die in diesem Abschnitt enthaltenen Zellen beschrieben.

## Die Model-Handler-Klasse

Die Klasse „Model-Handler“ wird verwendet, um das Training von Klassifizierungs-KNNs durchzuführen und die anfallenden Testergebnisse während des Trainings zu verwalten. Eine Instanz der Klasse Model-Handler bezieht sich immer auf ein Klassifizierungs-KNN. Für dieses Klassifizierungs-KNN werden die Loss-Werte vom Training sowie die durchschnittlichen Loss-Werte und Vorhersage-Genauigkeiten von den Tests auf den verwendeten Traings-, Validierungs- und Testdatensätzen festgehalten. Alle diese Ergebnisse können gesammelt mit den Klassifizierungs-KNNs über die Model-Handler-Instanz gespeichert werden. Dabei wird die Hyperparameter-Konfiguration des Klassifizierungs-KNN in JavaScript Object Notation (JSON) in einer Datei gespeichert.

Um die Ergebnisse zu produzieren, muss das verwaltete Klassifizierungs-KNN mit der entsprechenden Methode der Model-Handler-Instanz trainiert werden. Diese Methode instanziert zuerst mehrere Dataloader-Objekte, die nachfolgend dazu verwendet werden, Daten für das Training und die Tests des Klassifizierungs-KNN bereitzustellen. Die Dataloader-Objekte werden mit den im Abschnitt „Datensätze“ erstellten Dataset-Objekten bestückt. Über die Angabe einer Batch-Größe kann zusätzlich gesteuert werden, wie viele Datensätze die Dataloader-Objekte auf einmal bereitstellen. Dabei ist es wichtig zwischen den Dataloader-Objekten für das Training und das Testen zu unterscheiden. Die in Unterkapitel 3.3 betrachtete Batch-Größe wird nur für das Training verwendet. Beim Testen wird die Batch-Größe 1 verwendet.

Es wird ein Dataloader-Objekt für das Training (Train-Dataloader) und ein Dataloader-Objekt für die Validierung (Validation-Dataloader) mit den entsprechenden Dataset-Objekten erstellt. Für das Testen werden zwei Dataloader-Objekte (Test-Dataloader, Train-Bs1-Dataloader) erzeugt. Der Test-Dataloader nutzt die Testdatensätze während der Train-Bs1-Dataloader die Trainingsdatensätze verwendet. Es werden zwei Dataloader-Objekte auf Basis des Train-Dataset benötigt, da die Batch-Größe eines Dataloader-Objekts nicht geändert werden kann.

Der Trainingsprozess läuft in Epochen ab und wird für die Klassifizierungs-KNNs der vorliegenden Studienarbeit 15 Mal wiederholt. Jede Epoche beginnt damit, dass das Klassifizierungs-KNN mit dem Train-Dataloader trainiert wird. Anschließend werden

Tests mit dem Validation-, Test- und Train-Bs1-Dataloader durchgeführt und die Ergebnisse gespeichert.

Die Tests werden mit einer eigenen Methode durchgeführt. Während des gesamten Testprozesses wird das verwendete Klassifizierungs-KNN in einen Modus versetzt, in dem die Gewichte nicht angepasst werden. Dadurch bleiben die Validierungs- und Testdatensätze dem Klassifizierungs-KNN unbekannt, auch wenn die Datensätze das Klassifizierungs-KNN mehrfach durchlaufen haben. Für das Testen werden die Datensätze einzeln verarbeitet und die Ergebnisse über den gesamten Testprozess aggregiert.

Um den Trainingsprozess zu dokumentieren, enthält die Klasse Model-Handler zudem eine Methode, mit der die Trainings-, Validierungs- und Testergebnisse ausgegeben werden können.

### **Anstoßen der Trainings- und Test-Prozesse**

Der letzte Bestandteil des beschriebenen Jupyter Notebooks hängt stark von dem aktuell betrachteten Anwendungsfall ab. In diesem Teil des Jupyter Notebooks werden die Klassifizierungs-KNNs instanziert und einer Model-Handler-Instanz übergeben. Mit dieser Model-Handler-Instanz wird dann das Training des Klassifizierungs-KNN angestoßen.

# 5. Ergebnisse

Das fünfte Kapitel stellt die Ergebnisse der vorliegenden Studienarbeit vor. In dem nachfolgenden Kapitel 6 werden diese Ergebnisse interpretiert. In Unterkapitel 5.1 wird zuerst das Vorgehen für die Erhebung und den Vergleich der Testergebnisse beschrieben. Unterkapitel 5.2 geht näher auf die mit GANs erzeugten Bilder ein und vergleicht diese mit den echten Bildern. Das Unterkapitel 5.3 behandelt die Leistungen der auf verschiedenen Datensätzen trainierten Klassifizierungs-KNN.

## 5.1. Vorgehen und Bewertungs-Metriken

Die vorliegende Studienarbeit beschäftigt sich mit der Untersuchung des Effekts von synthetischen Bilddaten auf das Training von Klassifizierungs-KNNs. Diese Fragestellung wird beispielhaft an dem in Unterkapitel 2.5 beschriebenen Datensatz überprüft. Auf diesem Datensatz wurden die drei in Unterkapitel 3.2 beschriebenen GANs trainiert, um neue Datensätze für die gleiche Problemstellung zu erzeugen. In Unterkapitel 3.3 wurden drei Klassifizierungs-KNNs konzipiert, die separat mit echten, synthetischen und gemischten Datensätzen trainiert und getestet werden. Die Testergebnisse sind die Grundlage für die Beantwortung der betrachteten Fragestellung.

Die Testergebnisse bestehen aus den Vorhersage-Genauigkeiten und den durchschnittlichen Loss-Werten auf den Trainings-, Validierungs- und Testdatensätzen. Es werden in jedem Fall echte Validierungs- und Testdatensätze verwendet, um den Effekt der unterschiedlichen Trainingsdaten vergleichen zu können. Die Trainingsdatensätze können ausschließlich aus echten oder synthetischen sowie einer Mischung aus echten und synthetischen Daten bestehen. Nachfolgend werden diese drei Fälle näher beschrieben.

### 1. Echte Trainingsdaten

Beim Training der Klassifizierungs-KNNs ausschließlich auf echten Trainingsdaten werden immer rund 10.000 Trainingsdatensätze verwendet. In diesem Fall müssen keine weiteren Parameter der Trainingsdaten betrachtet werden.

## 2. Synthetische Trainingsdaten

Beim Training der Klassifizierungs-KNNs ausschließlich auf synthetischen Trainingsdaten können theoretisch unendlich viele Trainingsdatensätze verwendet werden. Das ist allerdings durch die begrenzenden Faktoren von Zeit und Rechenleistung nicht möglich. Im Rahmen der vorliegenden Studienarbeit wird das Training mit 10.000, 20.000 und 30.000 synthetischen Datensätzen betrachtet. Dabei muss zusätzlich nach dem GAN-Konzept unterschieden werden, das dem verwendeten Generator zu Grunde liegt (siehe Unterkapitel 5.2).

## 3. Gemischte Trainingsdaten

Beim Training der Klassifizierungs-KNNs auf einer Mischung aus echten und synthetischen Trainingsdaten werden alle 10.000 echten Trainingsdatensätze verwendet und um eine beliebige Anzahl synthetischer Datensätze ergänzt. Es gilt auch hier die gleiche Beschränkung durch die Faktoren Kosten und Rechenleistung. Daraus wird nur das Training von Klassifizierungs-KNNs mit 20.000 und 30.000 gemischten Datensätzen betrachtet. Auch hier muss nach dem GAN-Konzept unterschieden werden, das dem Generator zu Grunde liegt (siehe Unterkapitel 5.2).

In allen drei Fällen werden für alle Kombinationen der Datensatz-Parameter Instanzen der drei Klassifizierungs-KNN-Modelle trainiert und getestet. Alle Klassifizierungs-KNNs werden für 15 Epochen trainiert. Die Klassifizierungs-KNNs werden nach jeder Trainings-Epoche auf mithilfe der Trainings-, Validierungs- und den Testdaten getestet. Dabei werden der durchschnittliche Loss-Wert und die Vorhersage-Genauigkeit gemessen.

Der durchschnittliche Loss-Wert und die Vorhersage-Genauigkeit werden für die Bewertung der Leistung eines Klassifizierungs-KNNs herangezogen. In der vorliegenden Studienarbeit wird die Leistung von Klassifizierungs-KNNs primär durch die Vorhersage-Genauigkeit gemessen. Bei ähnlichen Vorhersagen bzw. Vorhersage-Genauigkeiten entscheidet der Loss-Wert über den Ausgang des Vergleichs. Allerdings muss beachtet werden, dass ein permanent schnell ansteigender Loss-Wert auch bei verhältnismäßig hohen Vorhersage-Genauigkeiten problematisch ist.

## 5.2. Synthetischen Daten

Bei der Konzeption der GANs in Unterkapitel 3.2 wurden drei unterschiedliche Fehlerfunktion in Betracht gezogenen. Die Generatoren der drei erstellten GANs werden entsprechend ihrer Fehlerfunktion nachfolgend folgendermaßen abgekürzt:

- Generator mit binärer Kreuzentropie: BCE-CGAN
- Generator mit quadratischem Fehler: LS-CGAN
- Generator mit Wasserstein-Distanz: WCGAN

Zuerst werden die mit dem BCE-CGAN generierten Bilder betrachtet, die in Abbildung 5.1 und 5.2 zu sehen sind. Optisch sehen die generierten Bilder den echten Bildern ähnlich. Der erzeugte Hintergrund ist wie bei den echten Bildern komplett schwarz. Die Zellen selber besitzen einen pink bis rötlichen Farbton, mit einzelnen lilafarbigen Stellen und dunklen Bereichen. Insbesondere der hell pinke Rand um jede Zelle ist auffällig. Bei den echten Zellbildern waren die Zellen hingegen einheitlich lila. Insgesamt kann dennoch gesagt werden, dass die generierten Zellen starke farbliche Ähnlichkeiten mit den echten Zellen aufweisen. In Hinblick auf die äußere Form der Zellen unterscheiden sich die synthetischen Bilder nur wenig von den echten Bildern. Während bei den echten Zellbildern die Unterschiede in der Zellform zwischen einer kranken und gesunden Zellen meist sehr gut zu erkennen waren, ist dies bei den generierten Zellen nicht der Fall. Unterschiede sind aus menschlicher Sicht kaum erkennbar. Es kann lediglich gesagt werden, dass die gesunden Zellen runder wirken als die kranken Zellen. Die kranken Zellen sind tendenziell unregelmäßiger.

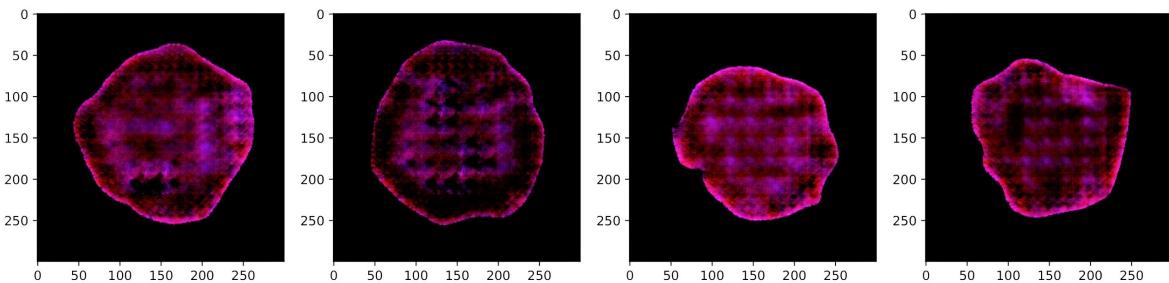


Abbildung 5.1.: Mit dem BCE-CGAN generierte Bilder von gesunden Zellen

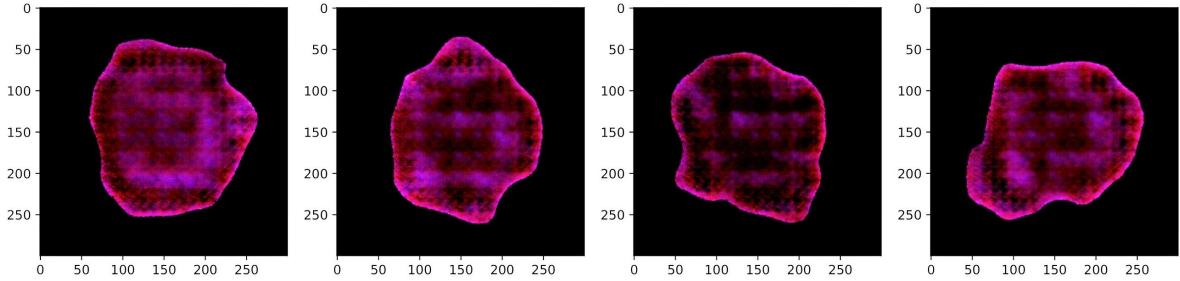


Abbildung 5.2.: Mit dem BCE-CGAN generierte Bilder von gesunden Zellen

Die mit dem LS-CGAN generierten Zellbilder besitzen genau wie die echten Zellbilder einen schwarzen Hintergrund. Bis auf den Hintergrund weichen die generierten Bilder jedoch stark von den echten Zellbildern ab. Die abgebildeten Zellen haben einen fast einheitlich pinken Farbton, der stärker von den echten Daten abweicht als es beim BCE-CGAN der Fall ist. Dazu kommt die unförmige und verschwommene Struktur der abgebildeten Zellen. Dadurch ähneln die mit dem BCE-CGAN erzeugten Bilder den echten Bildern nur wenig. Ein Unterschied zwischen gesunden und kranken Zellen ist ebenfalls nicht erkennbar. Die Abbildung 5.3 und 5.4 zeigen mit dem LS-CGAN generierte Bilder beider Klassen.

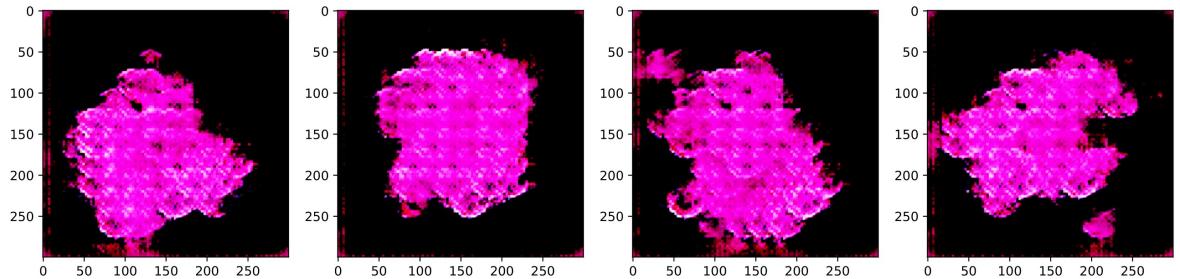


Abbildung 5.3.: Mit dem LS-CGAN generierte Bilder von gesunden Zellen

Auch die mit dem WCGAN generierten Bilder haben einen schwarzen Hintergrund. Die Farbe der generierten Zellen ist zum Großteil pink, wobei teilweise auch dunklere Stellen enthalten sind. Die Zellen in den mit dem WCGAN erzeugten Bildern weisen einen ähnlichen hell pinken Rand auf, wie schon bei den mit dem BCE-CGAN generierten Bildern. Es fällt auf, dass die dunkleren Stellen vor allem bei den kranken Zellen vorkommen und

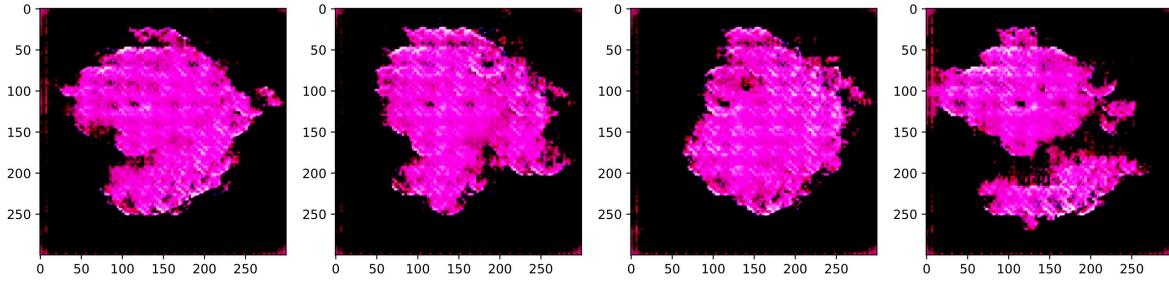


Abbildung 5.4.: Mit dem LS-CGAN generierte Bilder von kranken Zellen

die gesunden Zellen eine meist monotonere Farbgebung haben. Auf die Farbe reduziert ähneln die mit dem WCGAN generierten Bilder den echten Bildern ähnlich stark wie die mit dem BCE-CGAN generierten Bilder. In der Form unterscheiden sich kranke und gesunde Zellen bei den mit dem WCGAN generierten Bildern stärker als bei den mit dem BCE-CGAN generierten Bildern. Die Abbildung 5.5 und 5.6 zeigen mit dem WCGAN generierte Bilder beider Klassen.

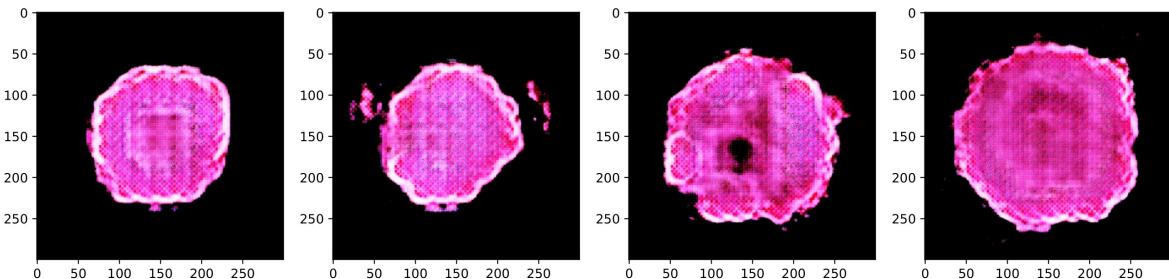


Abbildung 5.5.: Mit dem WCGAN generierte Bilder von gesunden Zellen

Aus menschlicher Sicht kann gesagt werden, dass die mit dem LS-CGAN generierten Bilder die schlechteste Qualität besitzen. Auf den generierten Bildern sind die Zellen nicht wirklich als solche zu erkennen. Die BCE-CGAN Zellbilder sehen den echten Zellbildern am ähnlichsten. Form und Farbe der generierten Bilder kommen hier den echten Bildern nahe. Die Beiden anderen GAN erreichen keine so starke Ähnlichkeit. Der fehlende Unterschied zwischen gesunden und kranken Zellen fällt bei den BCE-CGAN Bildern jedoch negativ auf. Es ist fraglich, ob ein Klassifizierungs-KNN zuverlässig zwischen gesunden und kranken Zellen unterscheiden kann, wenn Unterschiede in den Trainings-

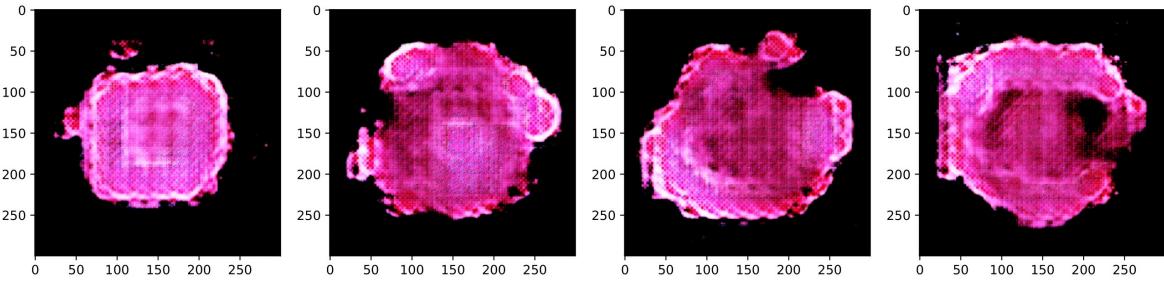


Abbildung 5.6.: Mit dem WCGAN generierte Bilder von kranken Zellen

daten menschlich nur schwer erkennbar sind. Insbesondere ist zu vermuten, dass dadurch eine schlechtere Generalisierung erreicht wird.

Die Bilder des WCGAN ähneln den echten Zellbildern, wenn auch nicht ganz so stark wie die Bilder des BCE-CGAN. Dafür weisen die generierten Bilder aber deutlich erkennbare Unterschiede zwischen kranken und gesunden Zellen auf. Diese Eigenschaft wirkt sich vermutlich positiv auf das Training von Klassifizierungs-KNNs aus.

Diese Erkenntnisse werden weitgehend von den FID-Scores der jeweiligen GANs unterstützt. Tabelle 5.1 zeigt für jedes GAN den FID-Score für einen generierten Datensatz mit gesunden Zellen und für einen Datensatz mit kranken Zellen. Es fällt auf, dass das BCE-CGAN für gesunde als auch kranke Zellen den mit Abstand besten FID-Score erreicht. Daraus lässt sich schließen, dass die BCE-CGAN Bilder den echten Zellbildern am ähnlichsten sind. Es ist erstaunlich, dass die Bilder des LS-CGAN einen etwas besseren FID-Score erreichen als die des WCGAN. Eine Erklärung dafür könnten die gut erkennbaren Unterschiede zwischen kranken und gesunden Zellen beim WCGAN sein. Das WGAN könnte gelernt haben, zu große Unterschiede zwischen kranken und gesunden Zellen zu erzeugen.

Auf Grundlage der eignen Beurteilung und der FID-Scores wird der Klassifizierer nur mit Bildern vom BCE-CGAN und dem WCGAN trainiert. Für die Wahl des BCE-CGAN spricht die Ähnlichkeit zu den echten Bildern und für die Wahl des WCGAN spricht die gute Unterscheidbarkeit zwischen gesunden und kranken Zellen. Ein Training mit Bildern des LS-CGAN ist aufgrund der schlechten Bildqualität nicht erfolgversprechend.

GAN	FID-Score gesunde Zellen	FID-Score kranke Zellen
BCE-CGAN	153	161
LS-CGAN	295	311
WCGAN	327	341

Tabelle 5.1.: Vergleich der FID-Scores für die unterschiedlichen GANs:

Das BCE-CGAN liefert mit Abstand den besten FID-Score. Das LS-CGAN erzielt die nächst besten FID-Scores. Am schlechtesten schneidet das WCGAN ab.

## 5.3. Testergebnisse der Klassifizierungs-KNNs

Dieses Unterkapitel beschreibt die Testergebnisse der in Unterkapitel 3.3 konzipierten Klassifizierungs-KNNs nach dem Training auf unterschiedlichen Trainings-Datensätzen. Die Parameter der Trainings-Datensätze wurden in Unterkapitel 5.1 und 5.2 beschrieben. Nachfolgend werden die besagten Testergebnisse dieser Klassifizierungs-KNNs auf den verwendeten Trainings-, Validierungs- und Testdaten separat betrachtet. Die erhobenen Testergebnisse werden dabei durch Abbildungen visualisiert. Jede Abbildung bezieht sich auf das Training mit bestimmten Datensätzen (z. B. 10.000 mit einem WCGAN-Generator generierte Datensätze) und zeigt die Vorhersage-Genauigkeit und den durchschnittlichen Loss-Wert auf den Trainings-, Validierungs- und Testdaten. Die in dem vorliegenden Unterkapitel referenzierten Abbildungen befinden sich in Anhang A.2.

### 5.3.1. Training mit echten Daten

Das Training mit echten Trainingsdaten ist der Ausgangspunkt für einen Vergleich mit synthetischen und gemischten Trainingsdaten. Denn die echten Trainingsdaten sind nicht mit Parametern wie der Datensatz-Anzahl oder dem Generator-Typ veränderbar. Abbildung A.6 zeigt die Testergebnisse eines auf echten Daten trainierten Klassifizierungs-KNN. Es ist zu erkennen, dass die Vorhersage-Genauigkeit auf den verwendeten echten Trainingsdaten für alle drei Klassifizierungs-KNN-Konzepte sehr schnell gegen 1 bzw. 100% konvergiert. Spätestens nach vier trainierten Epochen erzielen alle Klassifizierungs-KNNs bei dieser Betrachtung eine Vorhersage-Genauigkeit von 100% und erhalten diesen Wert bis zur 15ten Epoche. Damit einhergehend konvergiert der durchschnittliche Loss-Wert auf den verwendeten Trainingsdaten schnell gegen den Wert 0 und behält diesen

Wert bis zur 15ten Trainings-Epoche relativ konstant bei. Diese beiden Verläufe sind typisch für einen optimalen Trainingsfortschritt.

Die Testergebnisse auf den echten Validierungsdaten weisen einen ähnlichen Verlauf wie die Testergebnisse auf den Trainingsdaten auf (siehe Abbildung A.6). Allerdings konvergiert die Vorhersage-Genauigkeit bei dieser Betrachtung minimal unter 100% und der durchschnittliche Loss-Wert knapp über 0,1. Der durchschnittliche Loss-Wert des IncNet-Klassifizierungs-KNN steigt sogar nach der zwölften Trainings-Epoche wieder merkbar an.

Die Testergebnisse auf den echten Testdaten weisen bei der Vorhersage-Genauigkeit einen fast schon konstanten Verlauf bei ca. 80% über die Trainings-Epochen hinweg auf (siehe Abbildung A.6). Die Vorhersage-Genauigkeit auf den Testdaten liegt damit merkbar unter der Vorhersage-Genauigkeit auf den Validierungsdaten. Hinzu kommt, dass die Vorhersage-Genauigkeit wesentlich stärker schwankt und gegen Ende sogar abfällt. Der durchschnittliche Loss-Wert auf den Testdaten liegt nach der ersten Trainings-Epoche für alle Klassifizierungs-KNNs ungefähr bei 0,5 und steigt bis zur 15ten Trainings-Epoche unter stärkeren Schwankungen merkbar an. Besonders die EfficientNets erfahren eine große Steigerung des durchschnittlichen Loss-Werts.

### 5.3.2. Training mit synthetischen Daten

Die synthetischen Trainingsdaten können sich wie in Unterkapitel 5.1 und 5.2 aufgezeigt anhand der Parameter „Datensatz-Anzahl“ und „Generator-Typ“ unterscheiden. Diese Unterschiede der Trainingsdaten können sich auf die Leistungsfähigkeit der Klassifizierungs-KNNs auswirken. Um die unterschiedlichen Auswirkungen festzustellen, werden nachfolgend die Testergebnisse auf den Trainings-, Validierungs- und Testdaten für alle Kombinationen aus „Datensatz-Anzahl“ und „Generator-Typ“ separat betrachtet. Den nachfolgenden Ausführungen liegen die Abbildungen A.7 bis A.12 aus Anhang A.2 zu Grunde.

### Testergebnisse auf den Trainingsdaten

Die Testergebnisse auf den synthetischen Trainingsdaten unterscheiden sich nur minimal in Variation von Datensatz-Anzahl und Generator-Typ. Der Verlauf der Graphen für Vorhersage-Genauigkeit und durchschnittlichen Loss-Wert auf den Trainingsdaten (siehe Abbildungen A.7 bis A.12) ähnelt den Graphen für das Trainings mit echten Daten aus Abbildung A.6. Allerdings konvergieren die Graphen der Klassifizierungs-KNNs mit Training auf synthetischen Daten deutlich langsamer. Es fällt außerdem auf, dass die ResNets und IncNets bei der Wahl eines WCGAN-Generators etwas schneller konvergieren als bei einem BCE-CGAN-Generator.

### Testergebnisse auf den Validierungsdaten

Die Graphen der Testergebnisse auf den echten Validierungsdaten weisen erstmals einen völlig anderen als den optimalen Trainingsverlauf auf. Es zeigt sich, dass bei Verwendung eines BCE-CGAN-Generators die Vorhersage-Genauigkeit unabhängig von der Datensatz-Anzahl nicht langfristig steigt. Die Vorhersage-Genauigkeit schwankt bei der Verwendung eines BCE-CGAN Generators häufig stark, wodurch nur kurzfristige Steigerungen der Vorhersage-Genauigkeiten auftreten (siehe Abbildungen A.10 bis A.12). Besonders die ResNets sind unabhängig von der Datensatz-Anzahl anfällig für diese Schwankungen. Durch diese Schwankungen erzielen die Klassifizierungs-KNNs mehrmals vergleichsweise hohe Vorhersage-Genauigkeiten von ca. 70%. Auf der anderen Seite fallen die Vorhersage-Genauigkeiten dieser Klassifizierungs-KNNs schnell wieder unter 40%. Allgemein lässt sich sagen, dass die Vorhersage-Genauigkeit bei Verwendung eines BCE-CGAN-Generators hauptsächlich im Bereich von 40% bis 60% liegt (siehe Abbildungen A.10 bis A.12).

Bei Verwendung eines WCGAN schwankt die Vorhersage-Genauigkeit der Klassifizierungs-KNNs weniger als bei Verwendung eines BCE-CGAN-Generators (siehe Abbildungen A.7 bis A.9). Allerdings steigt auch hier die Vorhersage-Genauigkeit nicht langfristig, sondern bleibt gleich oder sinkt moderat ab. Stärkere längerfristig anhaltende Anstiege der Vorhersage-Genauigkeit treten nur vereinzelt auf (siehe z. B. IncNet in Abbildungen A.7).

Die Vorhersage-Genauigkeit hängt bei Verwendung eines WCGAN stärker von dem Klassifizierungs-KNN-Konzept ab, als bei dem BCE-CGAN-Generator. Bei 10.000 Trainingsdatensätzen befindet sich die Vorhersage-Genauigkeit der IncNets meist weit über 60%, während die EfficientNets meist ca. 50% und die ResNets unter 40% erzielen (siehe Abbildung A.7). Bei Verwendung von 20.000 und 30.000 Trainingsdatensätzen hingegen, erzielen die ResNets und EfficientNets meist Vorhersage-Genauigkeiten von mehr als 60%, während die IncNets nur selten die 50% überschreiten (siehe Abbildungen A.8 und A.9).

Der durchschnittliche Loss-Wert steigt unabhängig von der Datensatz-Anzahl und dem verwendeten Generator-Typ langfristig an (siehe Abbildungen A.7 bis A.12). Der Anstieg unterliegt dabei häufigen Schwankungen, wie sie auch bei den Vorhersage-Genauigkeiten zu beobachten sind. Die durchschnittlichen Loss-Werte auf den Validierungsdaten liegen für alle Klasifizierungs-KNNs nach der ersten Trainings-Epoche i. d. R. bei ca. 0,75. Von diesem Punkt steigen die durchschnittlichen Loss-Werte häufig stark an und erreicht Werte von bis zu 4.

Bei Verwendung eines BCE-CGAN-Generators führt ein Training mit 10.000 Datensätzen für alle Klassifizierugs-KNNs zu einer stetigen Steigerung des durchschnittlichen Loss-Werts (siehe Abbildung A.10). Auch beim Training mit 20.000 Datensätzen steigen die durchschnittlichen Loss-Werte an (siehe Abbildung A.11). Allerdings ist dieser Anstieg für die EfficientNets und auch die ResNets bei Bereinigung der Schwankung moderater. Lediglich der durchschnittliche Loss-Wert der IncNets steigt besonders stark an. Bei einem Training mit 30.000 Datensätzen steigt der durchschnittliche Loss-Wert für ResNets und IncNets nur moderat (siehe Abbildung A.12). Allerdings weisen dieses Mal die EfficientNets einen übermäßigen Anstieg des durchschnittlichen Loss-Werts auf.

Bei Verwendung eines WCGAN-Generators erzielen besonders die ResNets sehr hohe durchschnittliche Loss-Werte (siehe Abbildungen A.7 bis A.9). Die Efficientnets hingegen weisen unabhängig von der Datensatz-Anzahl einen sehr moderaten Anstieg des durchschnittlichen Loss-Werts auf. Die IncNets weisen bei 10.000 Trainingsdatensätzen die geringsten durchschnittlichen Loss-Werte auf (siehe Abbildung A.7). Bei 20.000 und 30.000 Trainingsdatensätzen steigt der durchschnittliche Loss-Wert der IncNets allerdings sporadisch über den Wert der ResNets (siehe Abbildungen A.8 und A.9).

### Testergebnisse auf den Testdaten

Die Testergebnisse bei den echten Testdaten unterscheiden sich unabhängig von der Datensatz-Anzahl und dem Generator-Typ nur minimal zu den Testergebnissen bei den echten Validierungsdaten (siehe Abbildungen A.7 bis A.12). Die Graphen der Vorhersage-Genauigkeit und des durchschnittlichen Loss-Werts weisen sehr ähnliche Verläufe auf. Es fällt auf, dass die Testergebnisse bei den Testdaten bei Verwendung eines BCE-CGAN-Generators etwas weniger schwanken als die Testergebnisse bei den Validierungsdaten (siehe Abbildungen A.10 bis A.12).

Bei Verwendung eines WCGAN stagniert die Vorhersage-Genauigkeit auf den Testdaten eher, als dass sie wie bei den Testergebnissen auf den Validierungsdaten abfällt (siehe Abbildungen A.7 bis A.9). Allgemein lässt sich sagen, dass die Vorhersage-Genauigkeit bei den Testdaten etwas geringer und der durchschnittliche Loss-Wert etwas größer ist als bei den Validierungsdaten.

### 5.3.3. Training mit gemischten Daten

In diesem letzten Abschnitt wird die Leistungsfähigkeit von Klassifizierungs-KNNs beschrieben, die mit einer Mischung aus echten und synthetischen Datensätzen trainiert wurden. Dazu wird auch hier wieder separat auf die Testergebnisse bei den Trainings-, Validierungs- und Testdaten eingegangen. Dabei werden durch die Parameter „Datensatz-Anzahl“ und „Generator-Typ“ entstandene Unterschiede in den Vorhersage-Genauigkeiten und den durchschnittlichen Loss-Werten hervorgehoben. Den nachfolgenden Ausführungen liegen die Abbildungen A.13 bis A.16 aus Anhang A.2 zu Grunde.

### Testergebnisse auf den Trainingsdaten

Die Graphen der Testergebnisse auf den gemischten Trainingsdaten weisen unabhängig von der Datensatz-Anzahl und dem Generator-Typ den optimalen Trainingsverlauf auf. Die Vorhersage-Genauigkeit konvergiert gegen 1 und der durchschnittliche Loss-Wert konvergiert gegen 0 (siehe Abbildungen A.13 bis A.16). Allerdings werden diese Grenzwerte meist erst nach der sechsten Trainings-Epoche erreicht.

### Testergebnisse auf den Validierungsdaten

Die Testergebnisse bei den echten Validierungsdaten zeigen unabhängig von der für das Training verwendeten Datensatz-Anzahl und dem Generator-Typ für alle Klassifizierungs-KNNs eine konstante Vorhersage-Genauigkeit von ca. 90%. Der zugehörige durchschnittliche Loss-Wert liegt meistens unter 0,25. Der durchschnittliche Loss-Wert steigt unabhängig von der Datensatz-Anzahl und dem Generator-Typ bei dieser Betrachtung nie über 0,6 (siehe Abbildungen A.13, A.16).

Bei einem Training mit 10.000 echten und 10.000 mit einem BCE-CGAN-Generator erzeugten Datensätzen sinkt der durchschnittliche Loss-Wert merkbar ab (siehe Abbildung A.15). Bei Erhöhung der Anzahl synthetischer Datensätze auf 20.000 steigt der durchschnittliche Loss-Wert mit zunehmender Anzahl Trainings-Epochen moderat an (siehe Abbildung A.16). Eine Ausnahme dafür sind die IncNets, die auch mit insgesamt 30.000 gemischten Trainingsdaten eine merkbare Reduktion des durchschnittlichen Loss-Werts auf den Validierungsdaten erreichen.

Bei Verwendung eines WCGAN-Generators sinkt der durchschnittliche Loss-Wert unabhängig von der Datensatz-Anzahl nach den ersten Trainings-Epochen merkbar ab (siehe Abbildungen A.13 und A.14). Bei Verwendung von insgesamt 20.000 gemischten Trainingsdatensätzen bleibt diese Reduktion des durchschnittlichen Loss-Werts langfristig erhalten. Eine Ausnahme davon sind die EficientNets. Bei Verwendung von insgesamt 30.000 gemischten Trainingsdatensätzen hingegen steigt der durchschnittliche Loss-Wert auf den Validierungsdaten bei den EfficientNets relativ schnell wieder an. Die ResNets erfahren schon nach der vierten Trainings-Epoche einen Anstieg des durchschnittlichen Loss-Werts, der bis zur 15ten Trainings-Epoche anhält und den durchschnittlichen Loss-Wert auf etwas weniger als 0,5 treibt. Die IncNets erfahren die kleinste Steigerung des durchschnittlichen Loss-Werts.

### Testergebnisse auf den Testdaten

Die Testergebnisse bei den echten Testdaten zeigen unabhängig von der für das Training verwendeten Datensatz-Anzahl und dem Generator-Typ für alle Klassifizierungs-KNNs eine konstante Vorhersage-Genauigkeit von etwas mehr als 75%. Dieses Verhalten ähnelt den Testergebnissen auf den Validierungsdaten.

Der durchschnittliche Loss-Wert verhält sich im Vergleich zu den Testergebnissen bei den Validierungsdaten anders. Denn unabhängig von der Datensatz-Anzahl und dem Generator-Typ steigt der durchschnittliche Loss-Wert auf den Testdaten seit der ersten Trainings-Epoche stark an (siehe Abbildungen A.13 bis A.16). Der durchschnittliche Loss-Wert liegt dabei nach der ersten Trainings-Epoche i. d. R. zwischen 0,5 und 0,6 und steigt häufig über 1. Starke Schwankungen des durchschnittlichen Loss-Werts auf den Testdaten führen unabhängig von der Datensatz-Anzahl und dem Generator-Typ zu Spitzenwerten von mehr als 2 (siehe Abbildung A.16). Diese Schwankungen treten nach der siebten Trainings-Epoche besonders stark in Erscheinung. Die stärksten Schwankungen treten bei allen Klassifizierungs-KNNs bei einem Training mit 10.000 echten und 20.000 mit einem WCGAN erzeugten Datensätzen auf (siehe Abbildung A.14).

Es fällt auf, dass die EfficientNets unabhängig von der Datensatz-Anzahl und dem Generator-Typ die höchsten durchschnittlichen Loss-Werte auf den Testdaten erzielen (siehe Abbildungen A.13 bis A.16). Da die durchschnittlichen Loss-Werte der EfficientNets jedoch stark schwanken, weisen die IncNets und ResNets meist geringere durchschnittliche Loss-Werte auf.

## 6. Diskussion und Bewertung

In dem vorliegenden Kapitel 6 werden die in Unterkapitel 5.3 geschilderten Testergebnisse der Klassifizierungs-KNNs interpretiert. Dabei wird besonders auf die Unterschiede zwischen einem Training mit echten, synthetischen und gemischten Datensätzen eingegangen. Um die nachfolgenden Ausführungen zu vereinfachen, werden auf echten Datensätzen trainierte Klassifizierungs-KNNs als „Real-Klassifizierungs-KNNs“ bezeichnet. Analog werden die Bezeichnungen „Syn-Klassifizierungs-KNN“ und „Mix-Klassifizierungs-KNN“ verwendet.

Die in Unterkapitel 5.3 beschriebenen Testergebnisse auf den jeweils verwendeten Trainingsdaten unterschieden sich nur minimal je nach Trainingsdatensatz. Alle Graphen der Testergebnisse auf den verwendeten Trainingsdaten weisen den für einen Trainingsfortschritt optimalen Verlauf auf. Das zeigt, dass in jedem Fall ein Trainingsfortschritt gemacht wird. Dass die Graphen der Testergebnisse auf den verwendeten Trainingsdaten einen optimalen Verlauf aufweisen, ist daher nicht verwunderlich. Denn während des Trainings wurden diese Trainingsdatensätze wiederholt verwendet, um die Gewichte der Klassifizierungs-KNNs anzupassen. Den Klassifizierungs-KNNs sind diese Datensätze somit bekannt.

Die Vorhersage-Genauigkeit und der durchschnittliche Loss-Wert der Syn-Klassifizierungs-KNNs konvergieren unabhängig von der verwendeten Datensatz-Anzahl und dem Generator-Typ am langsamsten gegen 1 bzw. 0. Zudem zeigt sich, dass die Syn-Klassifizierungs-KNNs nach der ersten Trainings-Epoche bis zur Konvergenz die geringsten Vorhersage-Genauigkeiten und die höchsten durchschnittlichen Loss-Werte aufweisen. Unabhängig von der verwendeten Datensatz-Anzahl und dem Generator-Typ erzielen die Mix-Klassifizierungs-KNNs bessere Testergebnisse auf den verwendeten Trainingsdaten. Es besteht bei der Vorhersage-Genauigkeit im Vergleich zu den Testergebnissen der Syn-Klassifizierungs-KNNs nur ein kleiner Unterschied. Allerdings besteht eine große Differenz bei den durchschnittlichen Loss-Werten.

Die mit Abstand besten Testergebnisse auf den verwendeten Trainingsdaten erzielen die Real-Klassifizierungs-KNNs. Die Real-Klassifizierungs-KNNs konvergieren sowohl bei

der Vorhersage-Genauigkeit als auch bei dem durchschnittlichen Loss-Wert wesentlich schneller als die Syn- und Mix-Klassifizierungs-KNNs. Hinzu kommt, dass die Real-Klassifizierungs-KNNs einer Vorhersage-Genauigkeit von 100% und einem durchschnittlichen Loss-Wert von 0 näher kommen, als die anderen Klassifizierungs-KNNs. Eine Vorhersage-Genauigkeit von nahezu 100% zusammen mit einem Loss-Wert nahe 0 deuten jedoch auch darauf hin, dass es zum Overfitting kommt und das KNN die Trainingsdaten auswendig lernt. Eine Folge davon kann eine schlechte Generalisierung des Modells sein.

Die Real-Klassifizierungs-KNNs weisen auch auf den echten Validierungsdaten sehr gute Testergebnisse auf. Die Vorhersage-Genauigkeit und der durchschnittliche Loss-Wert konvergieren zwar nicht mehr gegen die optimalen Werte (1 und 0). Allerdings ist eine Vorhersage-Genauigkeit von über 90% bei einem durchschnittlichen Loss-Wert von ca. 0,15 ein sehr gutes Ergebnis; besonders im Vergleich mit den Syn-Klassifizierungs-KNNs.

Denn die Syn-Klassifizierungs-KNNs erreichen unabhängig von der verwendeten Datensatz-Anzahl und dem Generator-Typ nie mehr als eine 75%ige Vorhersage-Genauigkeit auf den echten Validierungsdaten. Hinzu kommt, dass der durchschnittliche Loss-Wert der Syn-Klassifizierungs-KNNs in extreme Bereiche steigt und nicht absinkt. Das weist darauf hin, dass sich die echten Validierungsdaten stark von den synthetischen Trainingsdaten unterscheiden. Dadurch unterscheiden sich auch die beim Training auf den synthetischen Datensätzen extrahiert Merkmale von den relevanten Merkmalen der echten Validierungsdaten. Die Gewichte der Syn-Klassifizierungs-KNNs sind daher auf die Erkennung von zumindest teilweise anderen Merkmalen trainiert. Die Klassifizierung der echten Validierungsdaten mit den Syn-Klassifizierungs-KNNs kann somit nicht immer erfolgreich sein.

Die Testergebnisse der Mix-Klassifizierungs-KNNs auf den echten Validierungsdaten ähneln den Testergebnissen der Real-Klassifizierungs-KNNs. Die Vorhersage-Genauigkeit der Mix-Klassifizierungs-KNNs konvergiert unabhängig von der verwendeten Datensatz-Anzahl und dem Generator-Typ schnell gegen einen Wert über 90%. Der durchschnittliche Loss-Wert steigt bei den Mix-Klassifizierungs-KNNs in einigen Fällen stärker an. Allerdings bleibt der durchschnittliche Loss-Wert durchweg unter 0,5. In der Regel befinden sich die durchschnittlichen Loss-Werte in der Nähe von 0,25.

Diese Testergebnisse der Mix-Klassifizierungs-KNNs können auf ähnliche Weise wie die Testergebnisse der Syn-Klassifizierungs-KNNs erklärt werden. Die Mix-Klassifizierungs-

KNNs wurden mit allen echten und zusätzlichen synthetischen Datensätzen trainiert. Daher sind mindestens ein Großteil der in den echten Trainingsdaten enthaltenen Merkmale beim Training erlernt worden. Die Gewichte der Mix-Klassifizierungs-KNNs sind daher in der Lage einen großen Teil der Merkmale der echten Validierungsdaten zu erkennen und die Daten richtig zu klassifizieren. Bei einem Vergleich mit den durchschnittlichen Loss-Werten der Real-Klassifizierungs-KNNs fällt allerdings auf, dass die zusätzliche Verwendung von synthetischen Daten zu höheren Loss-Werten führt. Eine mögliche Erklärung für ein solches Ergebnis kann sein, dass die synthetischen Daten eher hinderlich sind und keinen Vorteil für die Klassifizierung bringen. Eine andere mögliche Erklärung ist, dass die synthetischen Daten zu einer besseren Generalisierung führen. Durch die teilweise unterschiedlichen Merkmale in den Trainingsdaten kann das KNN die Trainingsdaten nicht mehr auswendig lernen und kann die Klassifikationen deshalb nicht mehr mit der gleichen Sicherheit wie beim Training mit ausschließlich echten Daten vornehmen. Mit der zweiten Möglichkeit kann eine gleichbleibende Vorhersage-Genauigkeit bei leicht steigenden Loss-Werten erklärt werden, wie es in den Ergebnissen beobachtet wurde. Wären die synthetischen Trainingsdaten hinderlich, sollte auch eine Verschlechterung der Vorhersage-Genauigkeiten zu beobachten sein.

Die Vorhersage-Genauigkeiten der Real-Klassifizierungs-KNNs auf den echten Testdaten ist wesentlich niedriger als auf den Validierungsdaten und erreicht maximal etwas mehr als 80%. Auf den Testdaten ist auch keine echte Steigerung der Vorhersage-Genauigkeit durch das Training erkennbar. Der Loss-Wert steigt zudem stetig unter stärkeren Schwankungen an. Dieser extreme Unterschied der Testergebnisse auf den Validierungs- und den Testdaten ist sonderbar, denn die Validierungs- und die Testdaten bestehen beide ausschließlich aus echten Datensätzen und sind den Klassifizierungs-KNNs völlig unbekannt. Besonders der Unterschied bei den durchschnittlichen Loss-Werten macht stutzig. Denn die klassifizierungsrelevanten Merkmale der echten Validierungsdatensätze entsprechen scheinbar weitgehend den klassifizierungsrelevanten Merkmalen der echten Trainingsdaten. Die klassifizierungsrelevanten Merkmale der Testdaten hingegen scheinen nicht den klassifizierungsrelevanten Merkmalen der echten Trainingsdaten zu entsprechen. Denn der durchschnittliche Loss-Wert der gleichen Real-Klassifizierungs-KNNs sinkt auf den Validierungsdaten und steigt auf den Testdaten.

Nachforschungen haben ergeben, dass die Testdatensätze mit Bildern von Zellen anderer Patienten erstellt wurden als bei den Trainingsdaten. Die Validierungsdaten wurden vor

Beginn der ersten Trainings-Epoche aus den Trainingsdaten entnommen und von dort entfernt. Die Bilder der Validierungsdatensätze zeigen somit die Zellen der gleichen Patienten wie bei den Trainingsdaten. Diese Tatsache könnte die beschriebenen Unterschiede zwischen den Testergebnissen auf den Validierungs- und den Testdaten ausmachen. Das bedeutet, dass die vorhandenen echten Trainingsdatensätze hier nicht ausreichen, um eine echte Generalisierung zu erreichen.

Die Testergebnisse der Syn-Klassifizierungs-KNNs auf den echten Testdaten ähneln unabhängig von der verwendeten Datensatz-Anzahl und dem Generator-Typ den Testergebnissen auf den Validierungsdaten. Lediglich die Schwankungen sind auf den Testdaten weniger stark. Diese Ähnlichkeit zwischen den Testergebnissen auf den Validierungs- und den Testdaten kann bedeuten, dass die Syn-Klassifizierungs-KNNs besser generalisieren als die Real-Klassifizierungs-KNNs. Allerdings ist der durchschnittliche Loss-Wert der Syn-Klassifizierungs-KNNs auf den Validierungs- und Testdaten durchgängig zu hoch, als dass eine gute Generalisierung angenommen kann. Es ist wahrscheinlicher, dass die bei dem Training auf synthetischen Datensätzen gelernten Merkmale sich gleichermaßen von den klassifizierungsrelevanten Merkmalen der Validierungs- und Testdaten unterscheiden. Die Unterschiede zwischen den Testergebnissen auf den Validierungs- und Testdaten können daher kommen, dass die klassifizierungsrelevanten Merkmale der synthetischen Trainingsdaten den klassifizierungsrelevanten Merkmalen der echten Testdaten ähnlicher sind als denen der Validierungsdaten.

Die Testergebnisse der Mix-Klassifizierungs-KNNs auf den echten Testdaten ähneln stark den Testergebnissen der Real-Klassifizierungs-KNNs auf den Testdaten. Die Vorhersage-Genauigkeiten der Mix- und Real-Klassifizierungs-KNNs liegen sehr nah beieinander. Auch die durchschnittlichen Loss-Werte unterscheiden sich nur in seltenen Fällen stärker voneinander. In diesen seltenen Fällen erreichen die Mix-Klassifizierungs-KNNs höhere durchschnittliche Loss-Werte. Es lässt sich daher auch hier sagen, dass die zusätzliche Verwendung von synthetischen Trainingsdaten keinen besonderen Vorteil für die Klassifizierung bringt. Gleichzeitig wird die Klassifizierung aber auch nicht negativ beeinflusst durch die synthetischen Bilddaten.

Abschließend lässt sich sagen, dass die synthetischen Datensätze sowohl alleine als auch in Kombination mit echten Datensätzen für das Training der betrachteten Klassifizierungs-KNNs keinen nennenswerten Vorteil bringt. Es konnte allerdings festgestellt werden,

dass das Training mit von GANs generierten Trainingsdaten prinzipiell möglich ist und funktioniert. Lediglich die Qualität der generierten Bilder ist nicht ausreichend, um die Leistung von Klassifizierungs-KNNs im Vergleich zum Trainings mit echten Datensätzen zu erhöhen. Die klassifizierungsrelevanten Merkmale werden von den synthetischen Daten nicht vollständig abgebildet.

Auch das Verwenden einer Mischung aus echten und synthetischen Trainingsdaten führte nicht zu besseren Vorhersage-Genauigkeiten der Klassifizierungs-KNNs. Allerdings konnte beobachtet werden, dass die zusätzlichen synthetischen Daten zu einer verbesserten Generalisierungsfähigkeit der Klassifizierungs-KNNs führen. Der Einsatz von GANs zum Erzeugen synthetischer Trainingsdaten eignet sich Anfang 2022 somit für Data Augmentation und weniger als Ersatz für die echten Trainingsdaten.

Insgesamt muss beachtet werden, dass die in der vorliegenden Studienarbeit verwendeten Klassifizierungs-KNNs und GANs nicht dem professionellen Standard entsprechen. Die Leistung dieser Neuronalen Netze und die Qualität der generierten Bilder kann unter den richtigen Umständen möglicherweise so optimiert werden, dass die Ergebnisse sich unterscheiden.

## 7. Fazit und Ausblick

Das Ziel der vorliegenden Studienarbeit wurde erreicht. Es wurde am Beispiel von Bildern an Leukämie erkrankter Zellen gezeigt, inwiefern sich synthetische Daten für das Training von Klassifizierungs-KNNs eignen.

Zu Beginn der vorliegenden Studienarbeit wurden die grundlegenden Funktionsweisen von KNNs und CNNs für die Klassifizierung von Bilddaten erläutert. Anschließend wurden die GANs-Technologie und die aktuellen Probleme beim Training von GANs beleuchtet. In ersten Recherchen stellte sich heraus, dass das Training von Klassifizierungs-KNNs mit synthetischen Datensätzen möglich, aber schwierig ist. Die Recherche ergab zudem, dass eine Mischung aus echten und synthetischen Trainingsdaten erfolgversprechender ist und zu einer Leistungssteigerung für Klassifizierungs-KNNs führen kann.

Basierend auf diesen Erkenntnissen wurden GANs und Klassifizierungs-KNNs konzipiert. Die Grundlage für ein funktionierendes Training von KNNs ist eine gute Daten-Qualität. Damit die synthetischen Datensätze eine hohe Qualität haben, müssen diese den echten Daten möglichst ähnlich sehen. Die Qualität der generierten Bilder ist schwierig zu beurteilen. Es wurden mehrere GANs mit unterschiedlichen Loss-Funktionen trainiert. Die leistungsstärksten GANs wurden verwendet, um die synthetischen Datensätze für das Training von Klassifizierungs-KNNs zu erzeugen. Es wurden drei leistungsstarke Klassifizierungs-KNN-Konzepte (RestNet, InceptionNet und EfficientNet) ausgewählt und implementiert. Diese drei Klassifizierungs-KNNs wurden im Anschluss separat mit echten, synthetischen und gemischten Datensätzen trainiert.

Die Untersuchungsergebnisse der vorliegenden Studienarbeit bestätigen, dass das Training von Klassifizierungs-KNNs mit synthetischen Daten schwierig ist. Die Loss-Werte auf den Validierungs- und Testdaten steigen im Laufe des Trainings mit synthetischen Daten meist stark an und die Vorhersage-Genauigkeit bleibt verhältnismäßig niedrig. Die Leistung der auf echten Daten trainierten Klassifizierungs-KNNs war im Vergleich meist deutlich besser. Beim Training mit gemischten Daten zeigt sich eine etwas bessere Generalisierung der Klassifizierungs-KNNs bei vergleichbaren Leistungen. Allerdings blieb eine Steigerung der Vorhersage-Genauigkeit auf den Validierungs- und Testdaten aus.

Die Ergebnisse zeigen, dass mit GANs erzeugte Bilddaten prinzipiell für das Training von Klassifizierungs-KNNs geeignet sind. Insbesondere der Einsatz von synthetischen Daten zur Data Augmentation ist vielversprechend. Die erhoffte Leistungssteigerung bei der Klassifizierung durch den Einsatz synthetischer Daten blieb jedoch aus. Aufgrund der begrenzten Zeit und der limitierten Rechenleistung für die Entwicklung der GANs ist davon auszugehen, dass die verwendeten synthetischen Trainingsdaten noch weiter verbessert werden können. Durch leistungsstärkere GANs können qualitativ hochwertigere Bilddaten erzeugt werden, die möglicherweise zu einer Leistungssteigerung für Klassifizierungs-KNNs verwendet werden können. GANs sind Anfang des Jahres 2022 noch eine junge und wenig erforschte Technologie. Die GAN-Technologie wird daher noch nicht lange erforscht. Zukünftige Fortschritte in der GAN-Technologie könnten zu drastischen Verbesserungen der Qualität von synthetischen Daten führen. Die in der vorliegenden Studienarbeit betrachtete Fragestellung bleibt daher weiterhin aktuell.

# Literaturverzeichnis

- [1] Brock, O. „Künstliche Intelligenz und Robotik“. In: *Digitalisierung und Künstliche Intelligenz: Orientierungspunkte*. Hrsg. von Arnold Norbert; Wangermann, T. Berlin: Konrad-Adenauer-Stiftung e. V., 2018, S. 29–44.
- [2] Strecker, S. „Künstliche Neuronale Netze – Aufbau und Funktionsweise“. In: *Arbeitspapiere WI*. Bd. 10. Mainz: Lehrstuhl für Allg. BWL und Wirtschaftsinformatik, Johannes Gutenberg-Universität, 1997, S. 29–44.
- [3] Rashid, T. *Neuronale Netze selbst programmieren*. 1. Auflage. dpunkt.verlag GmbH, 2017.
- [4] Burkhardt, J. „Generierung von synthetischen Trainingsdaten für die Erkennung von Absenderdaten aus Brief-Korrespondenz“. Magisterarb. Universität Stuttgart – Institut für Maschinelle Sprachverarbeitung; Fraunhofer Institut für Arbeitswirtschaft und Organisation, 06/2020.
- [5] Aggerwal, C. *Neural Networks and Deep Learning*. 1. Auflage. Springer International Publishing AG, 2018.
- [6] Rashid, T. *GANs mit PyTorch selbst programmieren - Ein verständlicher Einstieg in Generative Adversarial Networks*. 1. Auflage. dpunkt.verlag, 2020.
- [7] Lanham, M. *Generating a New Reality: From Autoencoders and Adversarial Networks to Deepfakes*. 1. Auflage. Apress Media LLC, 2021.
- [8] Wiegand, A. *Eine Einführung in Generative Adversarial Network (GAN)*. Seminar. Universität Bamberg – Fakultät Wirtschaftsinformatik und Angewandte Informatik – Professur für Angewandte Informatik, insbes. Kognitive Systeme.
- [9] Abdusalamova, K. *Zelltypen des menschlichen Körpers*. 2021. URL: <https://www.kenhub.com/de/library/anatomie/zelltypen-des-menschlichen-körpers> (Einsichtnahme: 04. 10. 2021).
- [10] Dickten, H./ Kratsch, C./ Reiz, B. „Die künstliche Intelligenz in der Einzelzellgenomik“. In: *Gefäßchirurgie* 7 (09/2019), S. 523–530.

- [11] Meier, K. *Computer nach dem Vorbild des Gehirns?* 01/2007. URL: <https://www.uni-heidelberg.de/presse/rucu/rucu07-1/vorbild.html> (Einsichtnahme: 26. 10. 2021).
- [12] Kriesel, D. *Ein kleiner Überblick über Neuronale Netze.* 2007. URL: <http://www.dkriesel.com>.
- [13] Kornfeld, N. „Optimierung eines neuronalen Netzes zur Objekterkennung unter Verwendung evolutionärer Algorithmen“. Magisterarb. Freien Universität Berlin – Institut für Informatik, 04/2017.
- [14] Brown, T. B. u. a. *Language Models are Few-Shot Learners.* 05/2020. arXiv: 2005.14165 [cs.CL]. URL: <https://arxiv.org/pdf/2005.14165.pdf> (Einsichtnahme: 30. 11. 2021).
- [15] Hu, J. u. a. *Squeeze-and-Excitation Networks.* 05/2019. arXiv: 1709.01507 [cs.CV]. URL: <https://arxiv.org/pdf/1709.01507.pdf> (Einsichtnahme: 30. 11. 2021).
- [16] He, K. u. a. *Deep Residual Learning for Image Recognition.* 12/2015. arXiv: 1512.03385 [cs.CV]. URL: <https://arxiv.org/pdf/1512.03385.pdf> (Einsichtnahme: 30. 11. 2021).
- [17] Selle, S. *Künstliche Neuronale Netzwerke und Deep Learning.* Saarbrücken: Hochschule für Technik und Wirtschaft des Saarlandes – Fakultät für Wirtschaftswissenschaften – Professor für Wirtschaftsinformatik, 05/2018.
- [18] Scherer, A. *Künstliche Neuronale Netze.* Fernuniversität Hagen – Fakultät für Mathematik und Informatik. URL: [https://www.fernuni-hagen.de/mi/studium/module/pdf/Leseprobe-komplett\\_01834.pdf](https://www.fernuni-hagen.de/mi/studium/module/pdf/Leseprobe-komplett_01834.pdf) (Einsichtnahme: 12. 11. 2021).
- [19] Werbos, P. „Backpropagation through time: what it does and how to do it“. In: *Proceedings of the IEEE* 78.10 (1990), S. 1550–1560.
- [20] Folkers, A. *Steuerung eines autonomen Fahrzeugs durch Deep Reinforcement Learning.* Wiesbaden: Springer Fachmedien Wiesbaden, 2019. URL: [https://doi.org/10.1007/978-3-658-28886-0\\_2](https://doi.org/10.1007/978-3-658-28886-0_2) (Einsichtnahme: 12. 11. 2021).
- [21] Weuffel, D. „Einführung in Machine Learning“. Magisterarb. Geodätisches Institut der RWTH Aachen – FH Aachen, 01/2020.

- [22] Online, M. T. R. *Bilderkennung: Unschlagbar menschlich*. 11/2020. URL: <https://www.heise.de/hintergrund/Uncrackable-human-like-4226438.html> (Einsichtnahme: 05. 10. 2021).
- [23] Osman, A. *Maschinelles Lernen – Fortschrittliche Mustererkennung zur automatisierten Datenverarbeitung und Ergebnisanalyse*. 2016. URL: <https://www.izfp.fraunhofer.de/content/dam/izfp/de/documents/2017/Produkt-Maschinelles-Lernen.pdf> (Einsichtnahme: 03. 12. 2021).
- [24] Ebert, D. „Bildklassifizierung mit Neuronalen Netzen“. Magisterarb. Vitzenburg: Hochschule Merseburg, 04/2019.
- [25] Brunner, A. „Lernen von Aufnahmeparametern für CT-Messungen“. Magisterarb. Passau: Universität Passau – Fakultät für Informatik und Mathematik – Institut für Softwaresysteme in technischen Anwendungen der Informatik, 09/2019.
- [26] Prechelt, L. *Proben1 / A Set of Neural Network Benchmark Problems and Benchmarking Rules*. Techn. Ber. 21. Karlsruhe: Universität Karlsruhe – Fakultät für Informatik, 09/1994.
- [27] Bishop, C. *Neural Networks for Pattern Recognition*. Oxford: Clarendon Press, 1995.
- [28] Wurm, C. *Neuronale Netze und Tiefe Architekturen*. Techn. Ber. Düsseldorf: Heinrich Heine Universität Düsseldorf, 02/2021.
- [29] Von Stackelberg, B. „Konstruktionsverfahren vorwärtsgerichteter neuronaler Netze“. Diss. Stuttgart: Universität Stuttgart – Fakultät Mathematik und Physik – Institut für theoretische Physik, 2003.
- [30] Dittrich, F. „Künstliche neuronale Netze zur Verarbeitung natürlicher Sprache“. Bachelorarbeit. Leipzig: Hochschule für Technik, Wirtschaft und Kultur Leipzig – Fakultät Informatik und Medien, 02/2021.
- [31] Roberts, D. A./ Yaida, S./ Hanin, B. *The Principles of Deep Learning Theory*. 2021. arXiv: 2106.10165 [cs.LG].
- [32] Armbruster, A. „Kausalität“. In: *Frankfurter Allgemeine Zeitung* 7 (01/2022), S. 21.

- [33] Schrimpf, M. *Neuronale Netze - Eine Einführung in Lernverfahren*. Seminar Kognitive Robotik. Technische Universität München – Fakultät für Informatik – Forschungs- und Lehreinheit Informatik VI, 07/2012.
- [34] Brandenbusch, K. „Semantische Segmentierung mit Deep Convolutional Neural Networks“. Magisterarb. Technische Universität Dortmund – Fakultät für Informatik –, 11/2018.
- [35] Gevantmakher Mikhail; Meinel, C. *Medizinische Bildverarbeitung – Eine Übersicht*. Forschungsbericht 03. Universität Trier – Fachbereich IV Informatik – Institut für Telematik, 2004.
- [36] Goodfellow, I. J. u. a. *Generative Adversarial Networks*. 2014. arXiv: 1406.2661 [stat.ML].
- [37] Hui, J. *GAN — Why it is so hard to train Generative Adversarial Networks!* 2018. URL: <https://jonathan-hui.medium.com/gan-why-it-is-so-hard-to-train-generative-advisory-networks-819a86b3750b> (Einsichtnahme: 06. 11. 2021).
- [38] Arora, S./ Zhang, Y. *Do GANs actually learn the distribution? An empirical study*. 2017. arXiv: 1706.08224 [cs.LG].
- [39] Rashid, T. *Neuronale Netze selbst programmieren*. 1. Auflage. O'Reilly, 2017.
- [40] Arjovsky, M./ Chintala, S./ Bottou, L. *Wasserstein GAN*. 2017. arXiv: 1701.07875 [stat.ML].
- [41] Hui, J. *GAN — Wasserstein GAN & WGAN-GP*. 2018. URL: <https://jonathan-hui.medium.com/gan-wasserstein-gan-wgan-gp-6a1a2aa1b490> (Einsichtnahme: 25. 11. 2021).
- [42] Ba, H. *Improving Detection of Credit Card Fraudulent Transactions using Generative Adversarial Networks*. 2019. arXiv: 1907.03355 [cs.LG].
- [43] Santos Tanaka, F. H. K. dos/ Aranha, C. *Data Augmentation Using GANs*. 2019. arXiv: 1904.09135 [cs.LG].
- [44] Eilertsen, G. u. a. *Ensembles of GANs for synthetic training data generation*. 2021. arXiv: 2104.11797 [cs.CV].
- [45] Frid-Adar, M. u. a. „Synthetic data augmentation using GAN for improved liver lesion classification“. In: *2018 IEEE 15th International Symposium on Biomedical Imaging (ISBI 2018)*. 2018, S. 289–293.

- [46] Qin, Z. u. a. „A GAN-based image synthesis method for skin lesion classification“. In: *Computer Methods and Programs in Biomedicine* 195 (2020), S. 105568. URL: <https://www.sciencedirect.com/science/article/pii/S0169260720302418>.
- [47] Mourya, S. u. a. *ALL Challenge dataset of ISBI 2019*. 2019.
- [48] Brownlee, J. *How to implement the Frechet Inception Distance (FID) for evaluating Gans*. 10/2019. URL: <https://machinelearningmastery.com/how-to-implement-the-frechet-inception-distance-fid-from-scratch/> (Einsichtnahme: 19.03.2022).
- [49] Gauthier, J. „Conditional generative adversarial nets for convolutional face generation“. In: (2015). URL: [http://cs231n.stanford.edu/reports/2015/pdfs/jgauthie\\_final\\_report.pdf](http://cs231n.stanford.edu/reports/2015/pdfs/jgauthie_final_report.pdf).
- [50] Brownlee, J. *How to develop a conditional gan (cgan) from scratch*. 09/2020. URL: <https://machinelearningmastery.com/how-to-develop-a-conditional-generative-adversarial-network-from-scratch/> (Einsichtnahme: 24.03.2022).
- [51] Mao, X./ Li, Q. *Generative Adversarial Networks for Image Generation*. 1. Auflage. Springer, Singapore, 2020.
- [52] Szegedy, C. u. a. *Going Deeper with Convolutions*. 2014. URL: <https://arxiv.org/abs/1409.4842>.
- [53] Szegedy, C. u. a. *Rethinking the Inception Architecture for Computer Vision*. 2015. URL: <https://arxiv.org/abs/1512.00567>.
- [54] Hüskens, M. „Evolution und Lernen zur Optimierung neuronaler Strukturen“. Diss. Ruhr-Universität Bochum – Fakultät für Physik und Astronomie, 06/2003.
- [55] Hanhart, M./ Stalz-John, D. „Darum hat sich Python zur Sprache für maschinelles Lernen gemausert“. In: *iX* 1 (2021), S. 46–49. URL: <https://www.heise.de/select/ix/2021/1/2026911212863502355>.
- [56] Müller, J./ Bachfeld, D. *Anaconda: Python und Jupyter installieren und bedienen*. 12/2021. URL: <https://www.heise.de/ratgeber/Anaconda-Python-und-Jupyter-installieren-und-bedienen-6276687.html?seite=all>.
- [57] jupyter.org. *Jupyter*. URL: <https://jupyter.org>.
- [58] O'Connor, R. *PyTorch vs TensorFlow in 2022*. 2021. URL: <https://www.assemblai.com/blog/pytorch-vs-tensorflow-in-2022> (Einsichtnahme: 28.02.2022).

- [59] He, H. *The State of Machine Learning Frameworks in 2019*. 2019. URL: <https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/> (Einsichtnahme: 28. 02. 2022).

# **A. Anhang**

## **A.1. Hyperparameter-Suche für Klassifizierungs-KNNs**

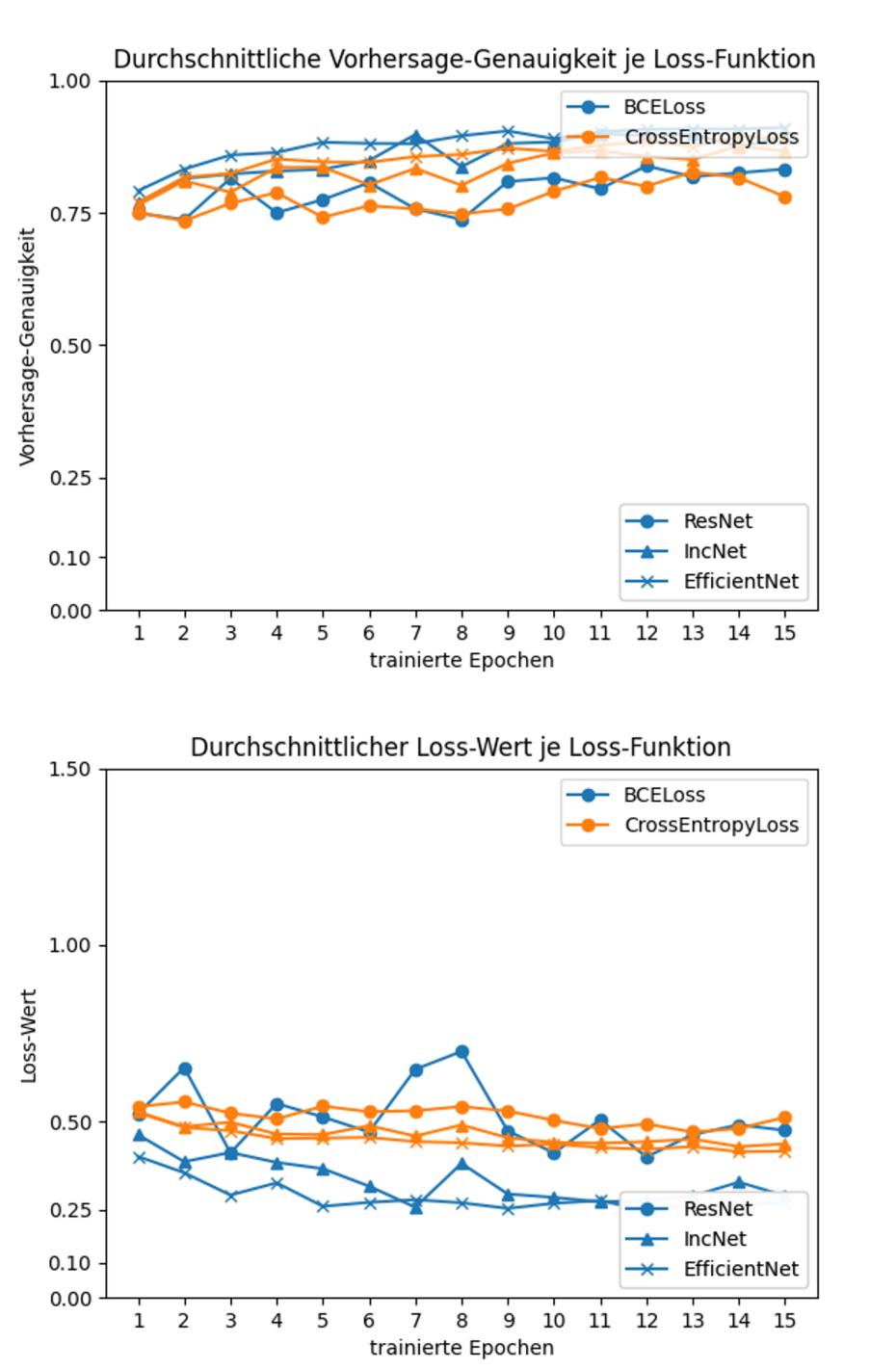


Abbildung A.1.: Vergleich der Loss-Funktionen für Klassifizierungs-KNNs:

Die Ergebnisse der Tests aus Kapitel 3.3 werden in dieser Abbildung über die Loss-Funktion (Legende oben rechts) und die Konzepte (Legende unten rechts) aggregiert dargestellt.

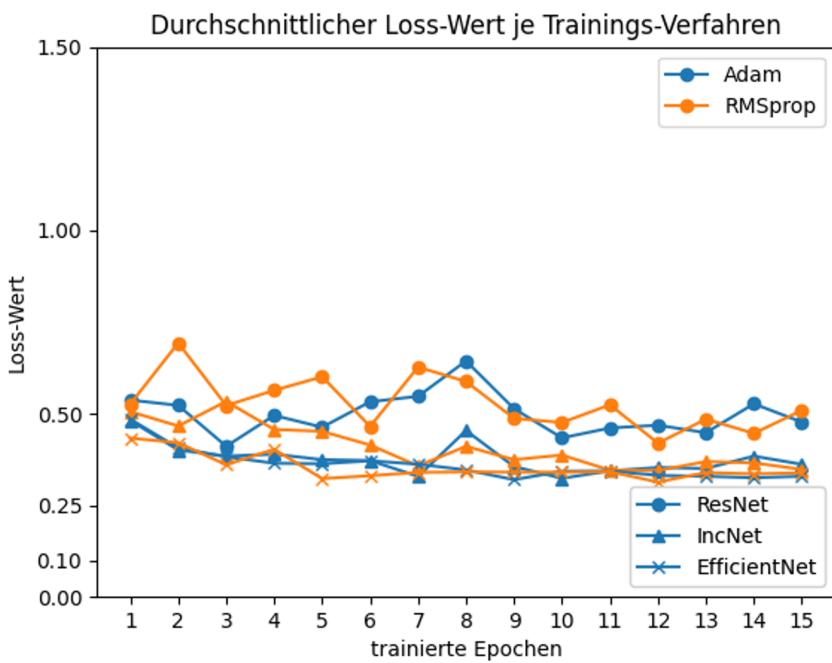
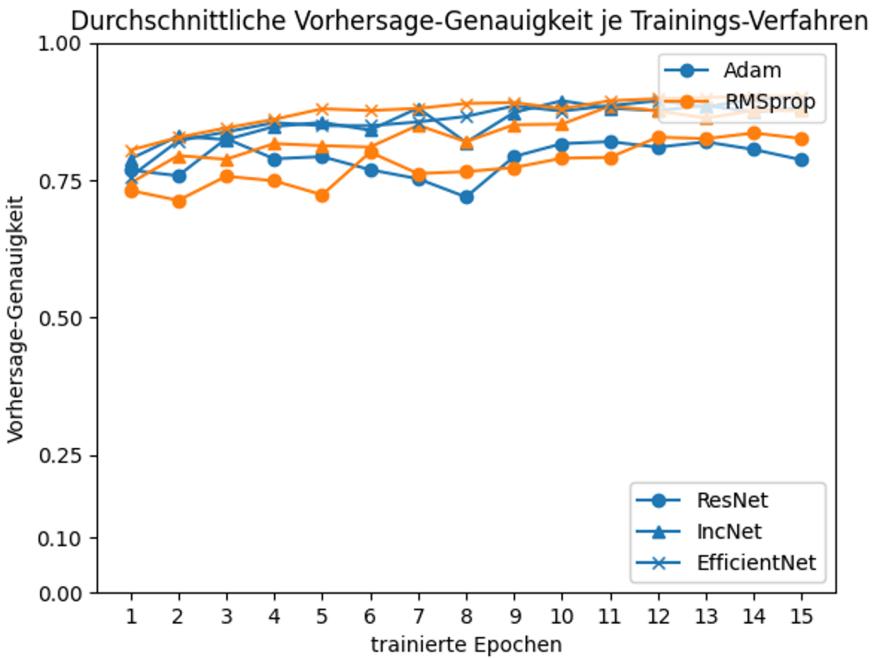


Abbildung A.2.: Vergleich der Trainings-Verfahren für Klassifizierungs-KNNs:

Die Ergebnisse der Tests aus Kapitel 3.3 werden in dieser Abbildung über das Trainings-Verfahren (Legende oben rechts) und die Konzepte (Legende unten rechts) aggregiert dargestellt.

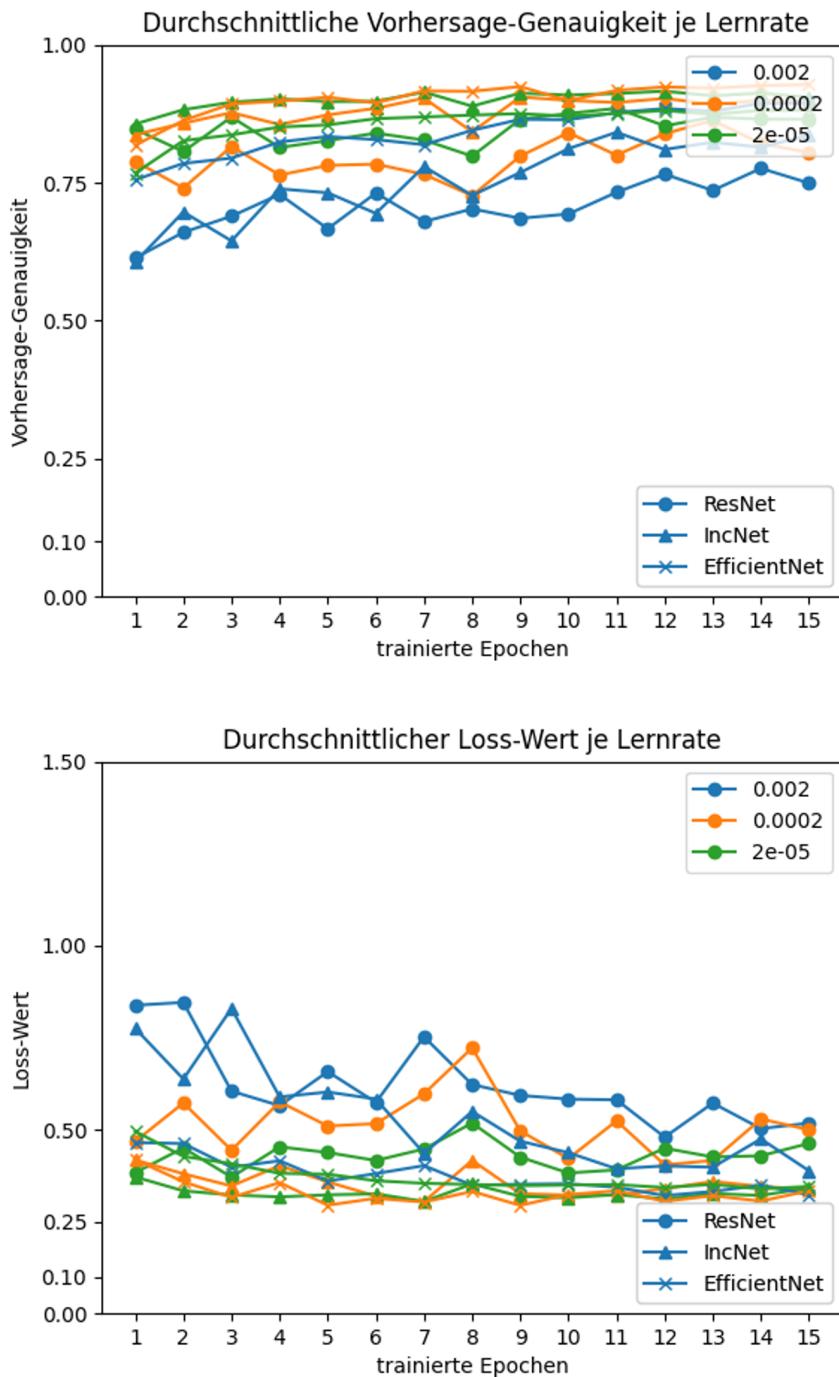


Abbildung A.3.: Vergleich der Lernraten für Klassifizierungs-KNNs:

Die Ergebnisse der Tests aus Kapitel 3.3 werden in dieser Abbildung über die Lernrate (Legende oben rechts) und die Konzepte (Legende unten rechts) aggregiert dargestellt.

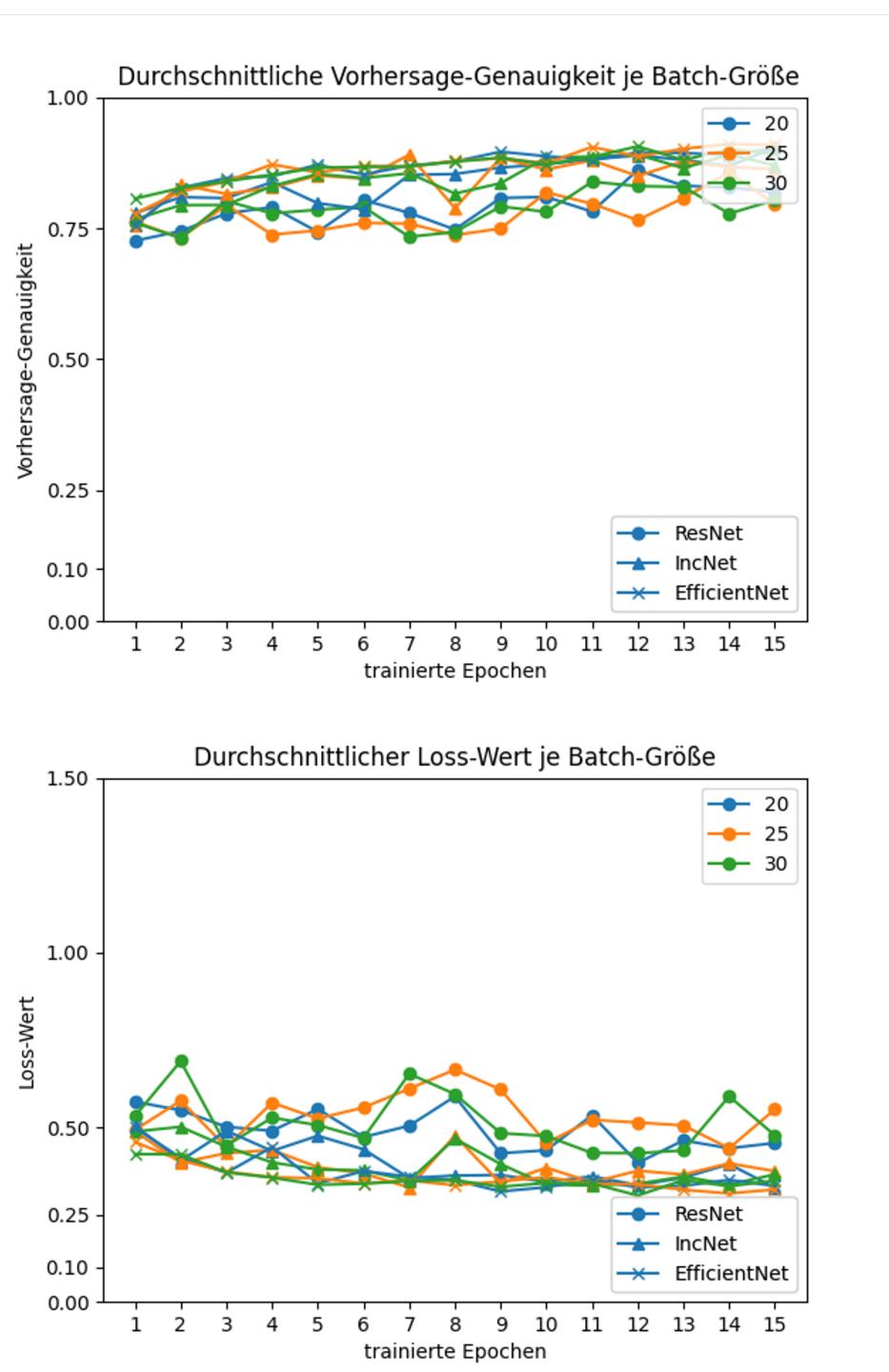


Abbildung A.4.: Vergleich der Batch-Größen für Klassifizierungs-KNNs:

Die Ergebnisse der Tests aus Kapitel 3.3 werden in dieser Abbildung über die Batch-Größe (Legende oben rechts) und die Konzepte (Legende unten rechts) aggregiert dargestellt.

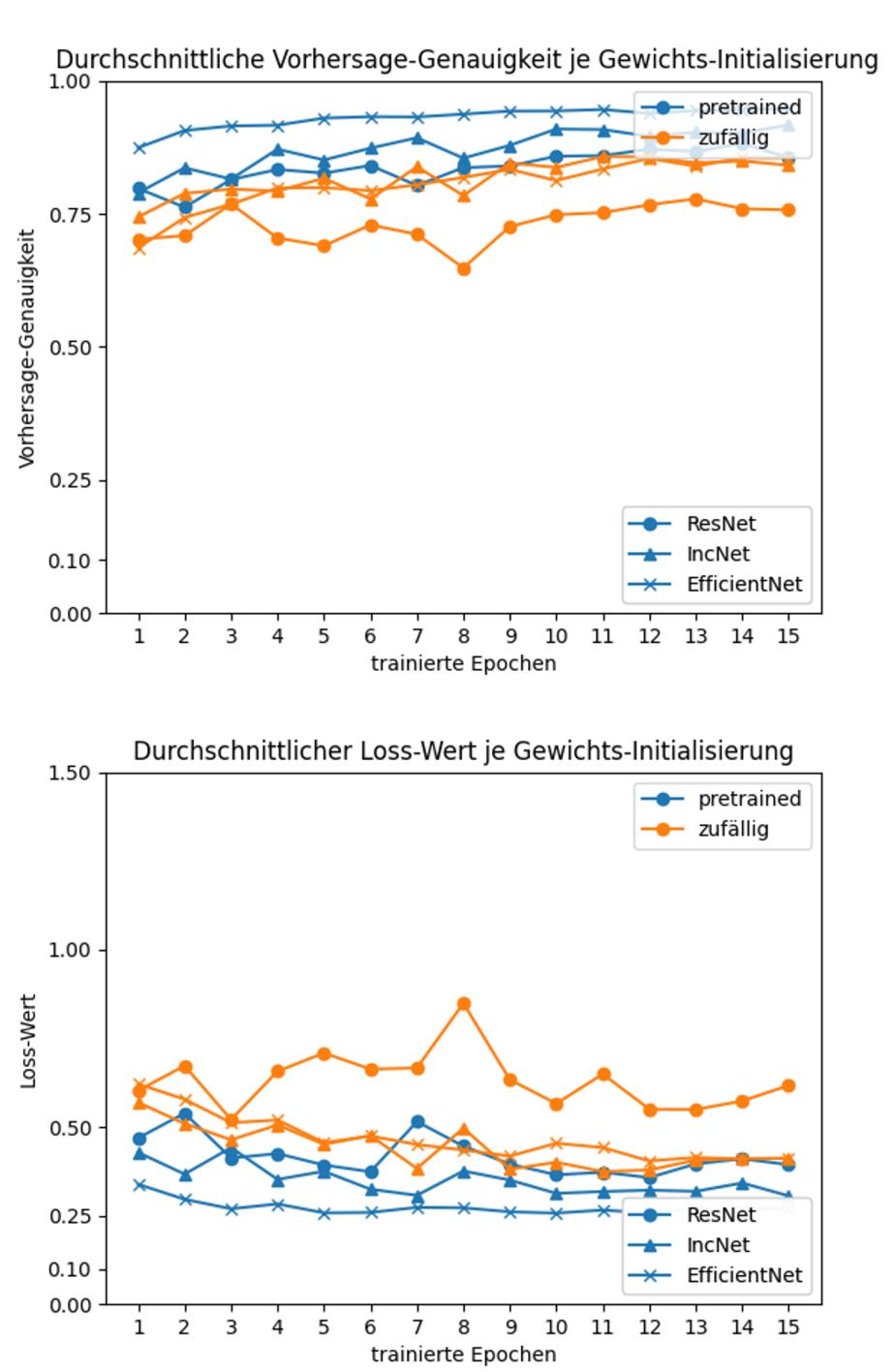


Abbildung A.5.: Vergleich der Gewichts-Initialisierung für Klassifizierungs-KNNs:  
Die Ergebnisse der Tests aus Kapitel 3.3 werden in dieser Abbildung über die Form der Gewichts-Initialisierung (Legende oben rechts) und die Konzepte (Legende unten rechts) aggregiert dargestellt.

## A.2. Testergebnisse der Klassifizierungs-KNNs

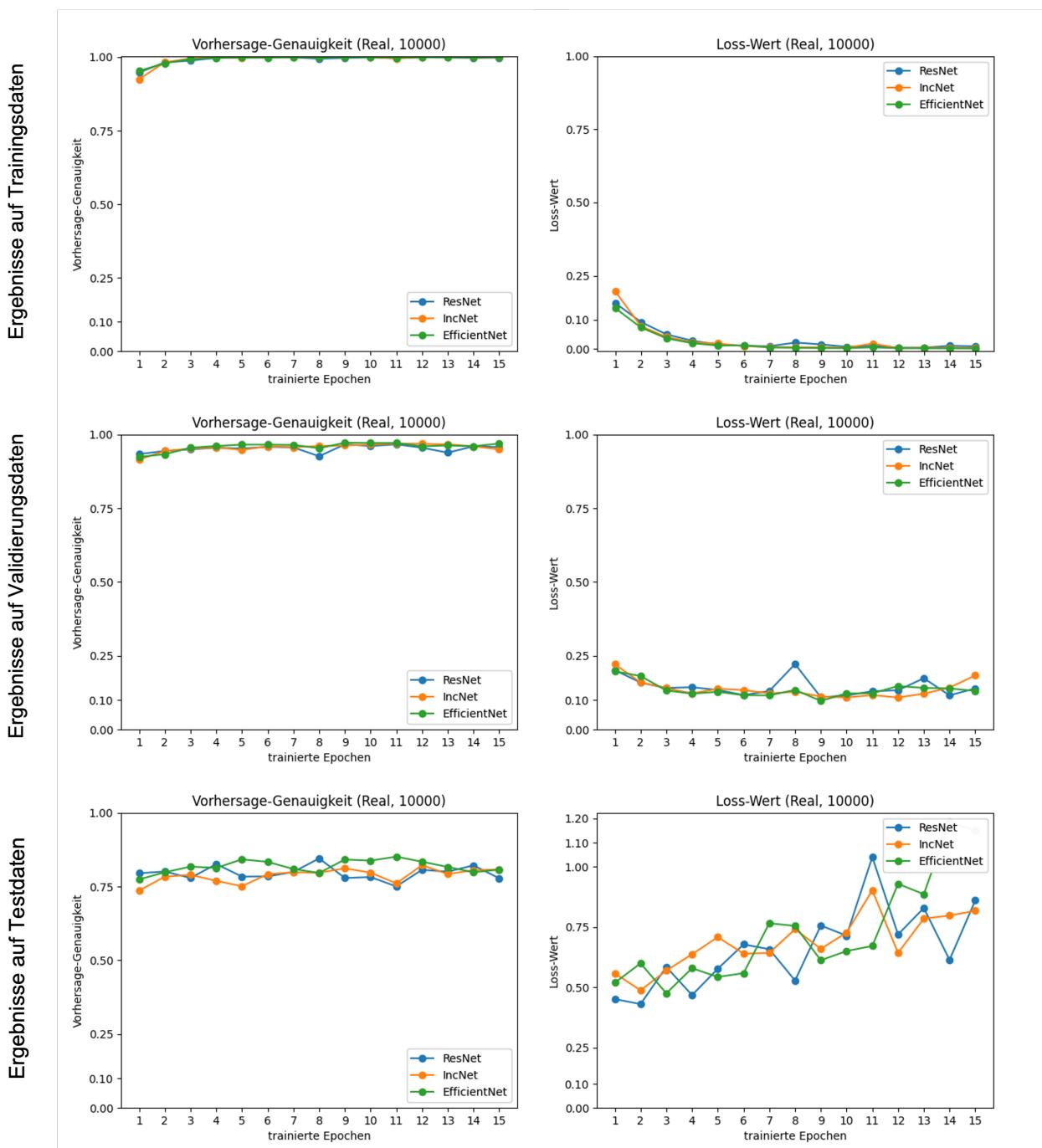


Abbildung A.6.: Testergebnisse eines auf echten Daten trainierten Klassifizierungs-KNN:  
Die Abbildung zeigt die Testergebnisse eines auf echten Trainingsdaten trainierten Klassifizierungs-KNN. Es werden die Vorhersage-Genauigkeit (linke Graphen) und der durchschnittliche Loss-Wert (rechte Graphen) für alle Klassifizierungs-KNN-Konzepte auf den verwendeten Trainingsdaten (oben) sowie den echten Validierungs- (mittig) und Testdaten (unten) abgebildet.

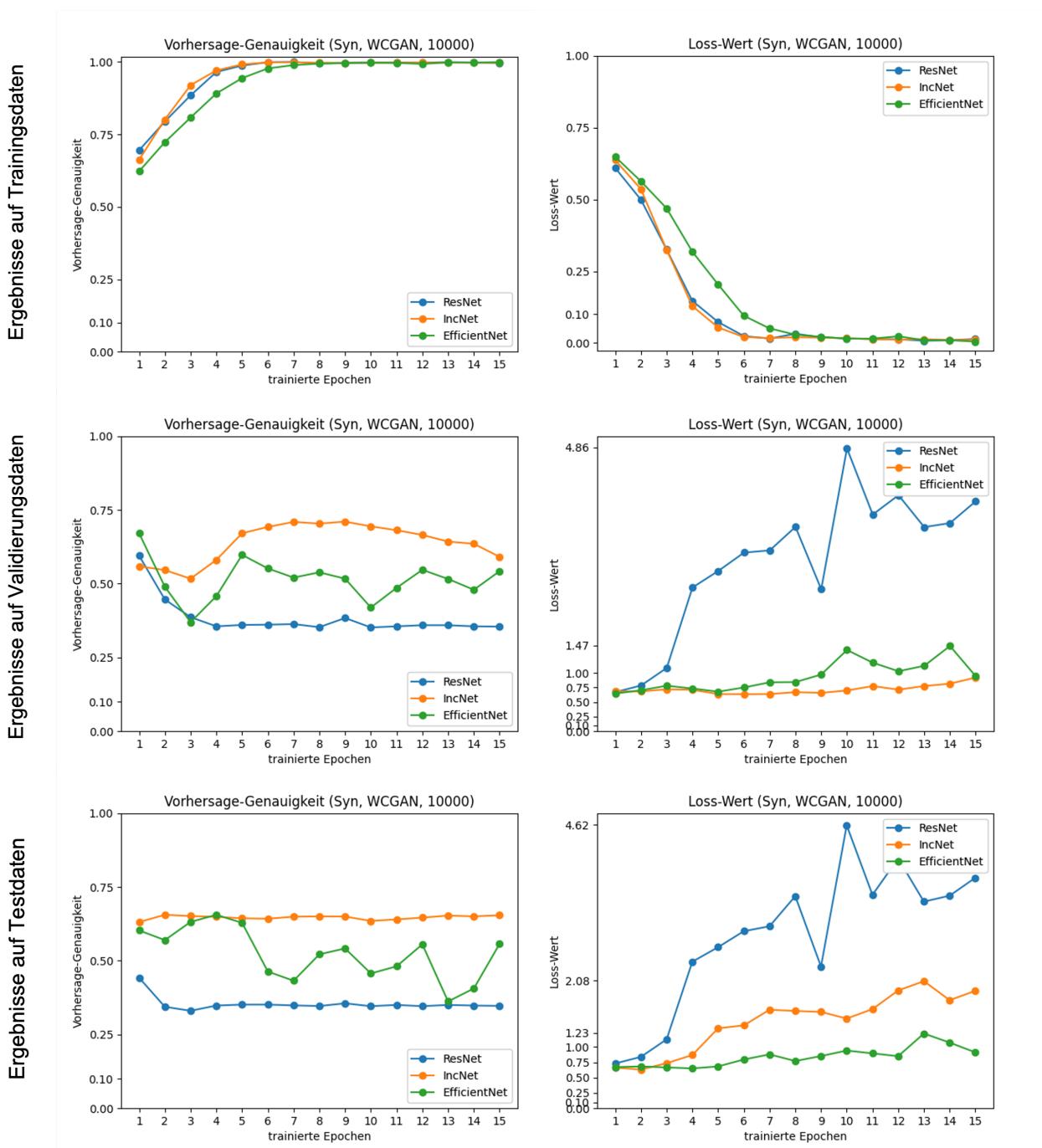


Abbildung A.7.: Testergebnisse eines auf 10.000 mit einem Wasserstein-CGAN generierten synthetischen Daten trainierten Klassifizierungs-KNN:  
Es werden die Vorhersage-Genauigkeit (linke Graphen) und der durchschnittliche Loss-Wert (rechte Graphen) für alle Klassifizierungs-KNN-Konzepte auf den verwendeten Trainingsdaten (oben) sowie den echten Validierungs- (mittig) und Testdaten (unten) abgebildet.

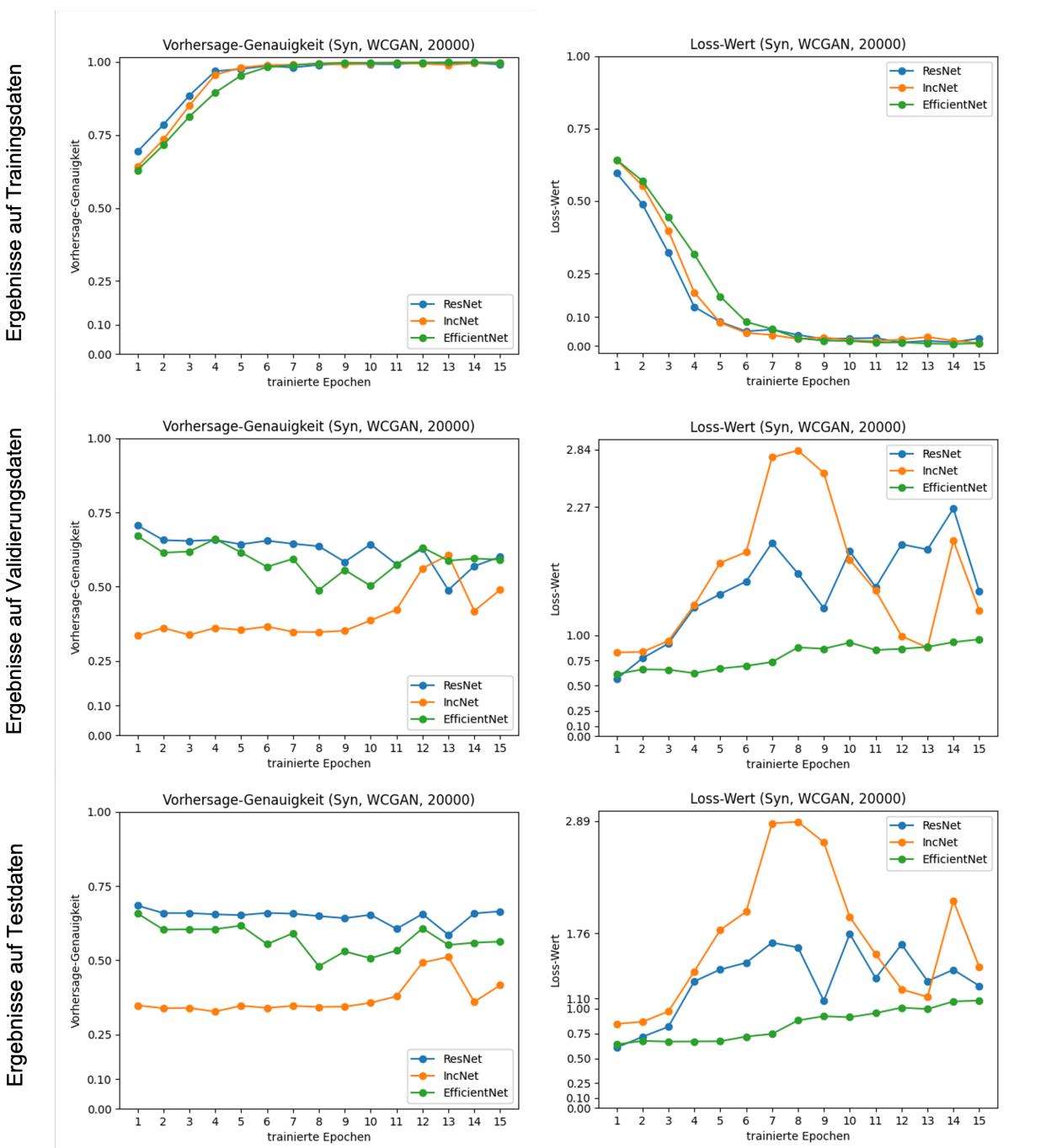


Abbildung A.8.: Testergebnisse eines auf 20.000 mit einem Wasserstein-CGAN generierten synthetischen Daten trainierten Klassifizierungs-KNN:  
Es werden die Vorhersage-Genauigkeit (linke Graphen) und der durchschnittliche Loss-Wert (rechte Graphen) für alle Klassifizierungs-KNN-Konzepte auf den verwendeten Trainingsdaten (oben) sowie den echten Validierungs- (mittig) und Testdaten (unten) abgebildet.

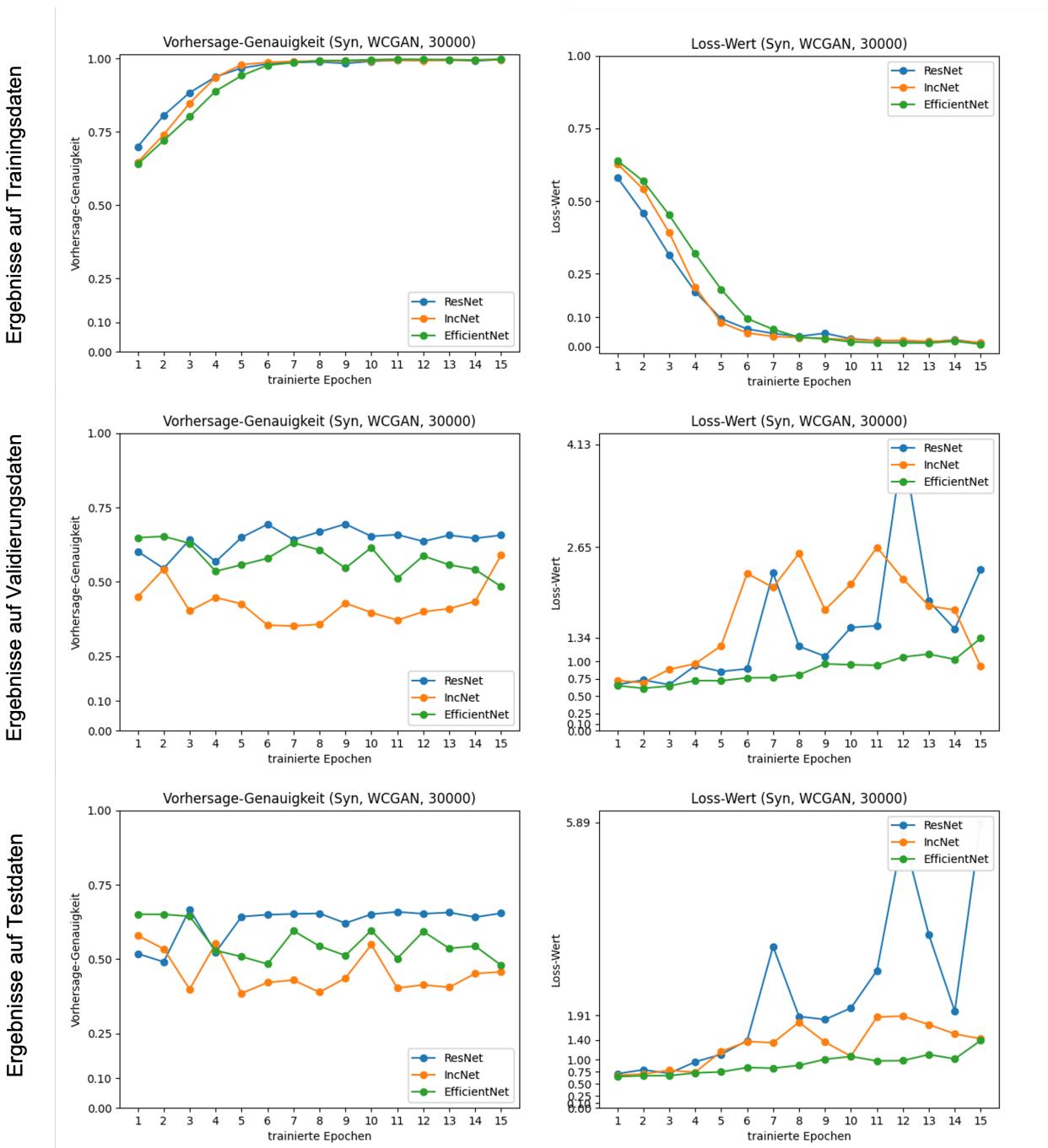


Abbildung A.9.: Testergebnisse eines auf 30.000 mit einem Wasserstein-CGAN generierten synthetischen Daten trainierten Klassifizierungs-KNN:  
Es werden die Vorhersage-Genauigkeit (linke Graphen) und der durchschnittliche Loss-Wert (rechte Graphen) für alle Klassifizierungs-KNN-Konzepte auf den verwendeten Trainingsdaten (oben) sowie den echten Validierungs- (mittig) und Testdaten (unten) abgebildet.

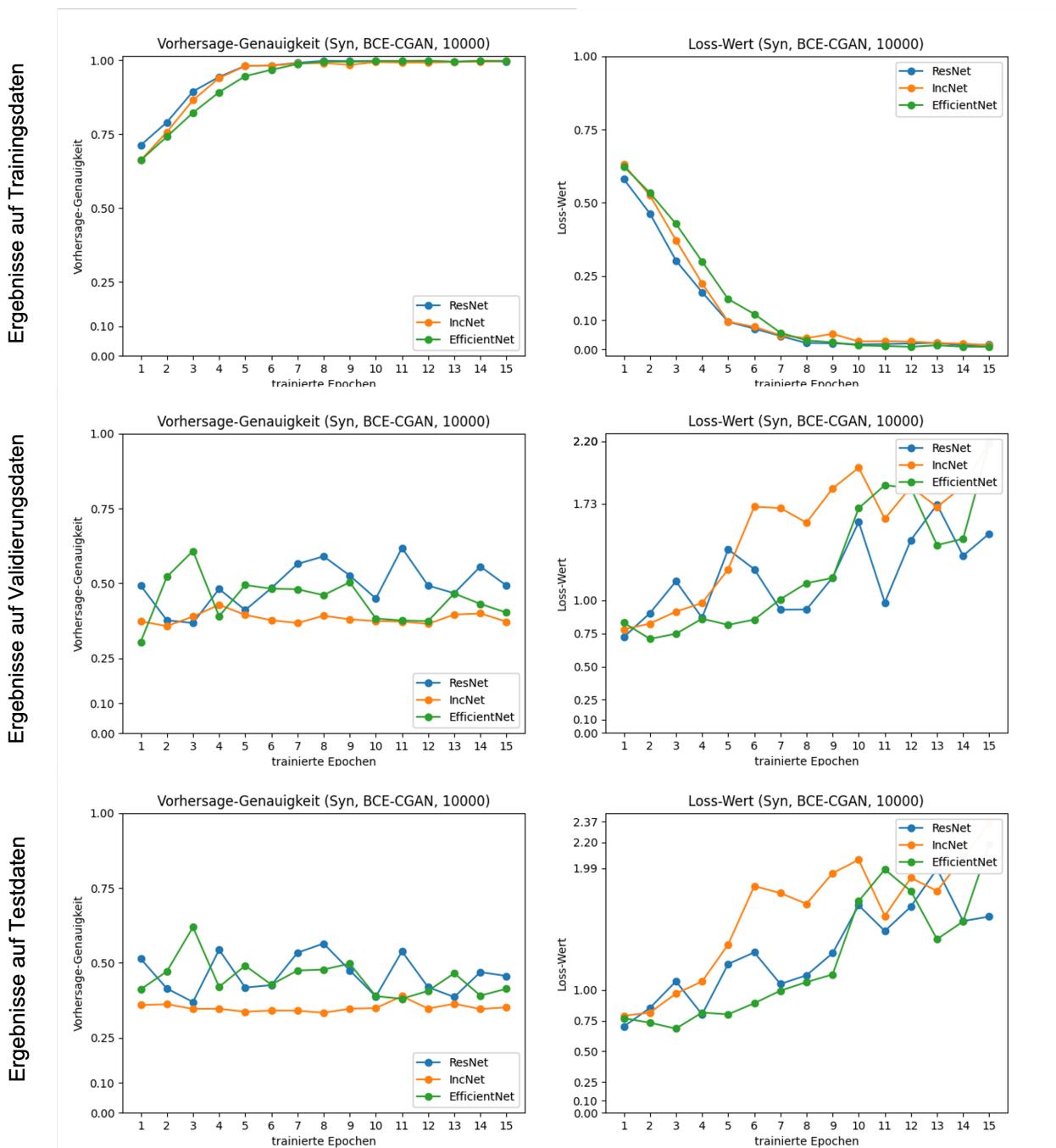


Abbildung A.10.: Testergebnisse eines auf 10.000 mit einem BCE-CGAN generierten synthetischen Daten trainierten Klassifizierungs-KNN:  
 Es werden die Vorhersage-Genauigkeit (linke Graphen) und der durchschnittliche Loss-Wert (rechte Graphen) für alle Klassifizierungs-KNN-Konzepte auf den verwendeten Trainingsdaten (oben) sowie den echten Validierungs- (mittig) und Testdaten (unten) abgebildet.

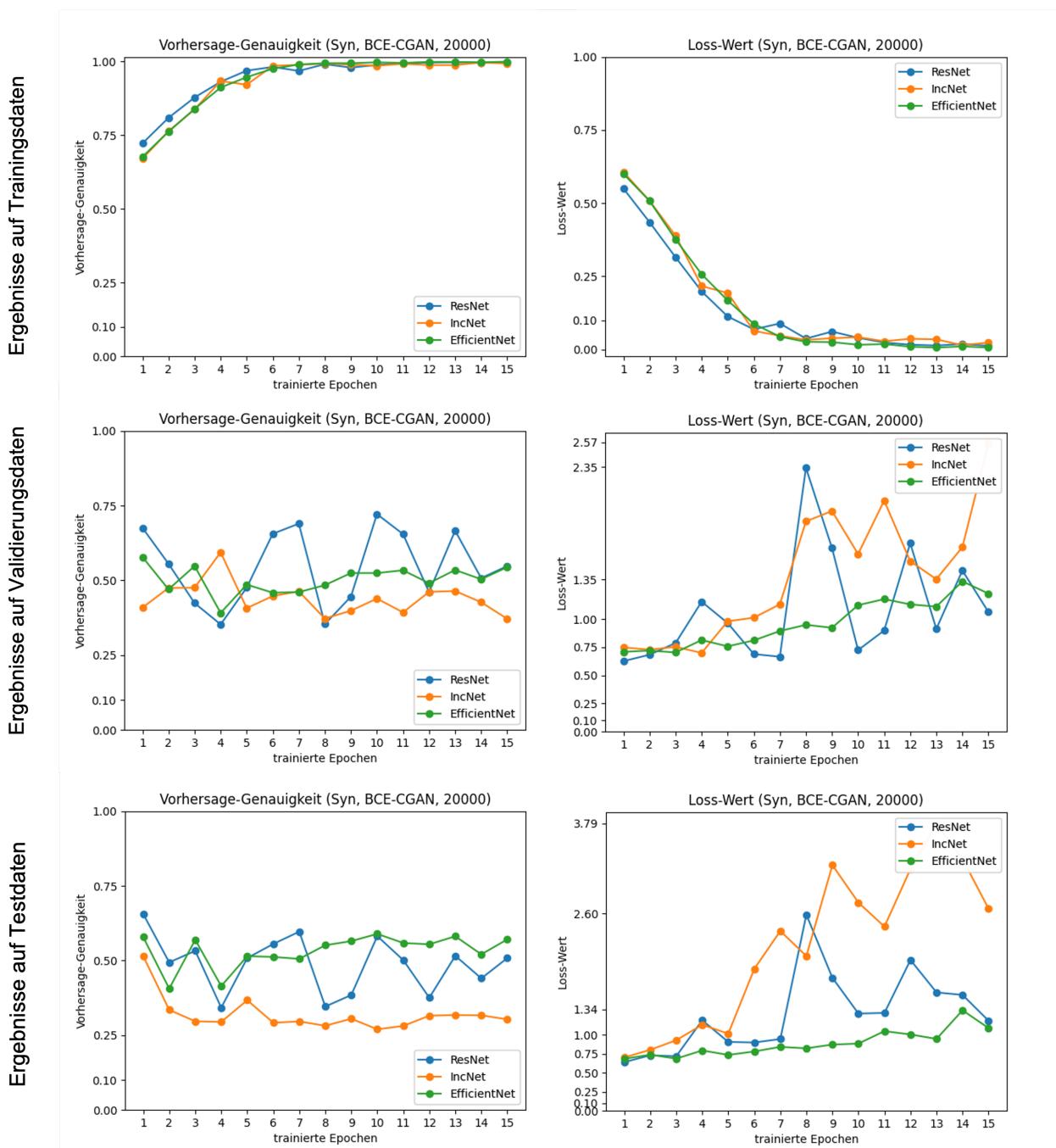


Abbildung A.11.: Testergebnisse eines auf 20.000 mit einem BCE-CGAN generierten synthetischen Daten trainierten Klassifizierungs-KNN:  
Es werden die Vorhersage-Genauigkeit (linke Graphen) und der durchschnittliche Loss-Wert (rechte Graphen) für alle Klassifizierungs-KNN-Konzepte auf den verwendeten Trainingsdaten (oben) sowie den echten Validierungs- (mittig) und Testdaten (unten) abgebildet.

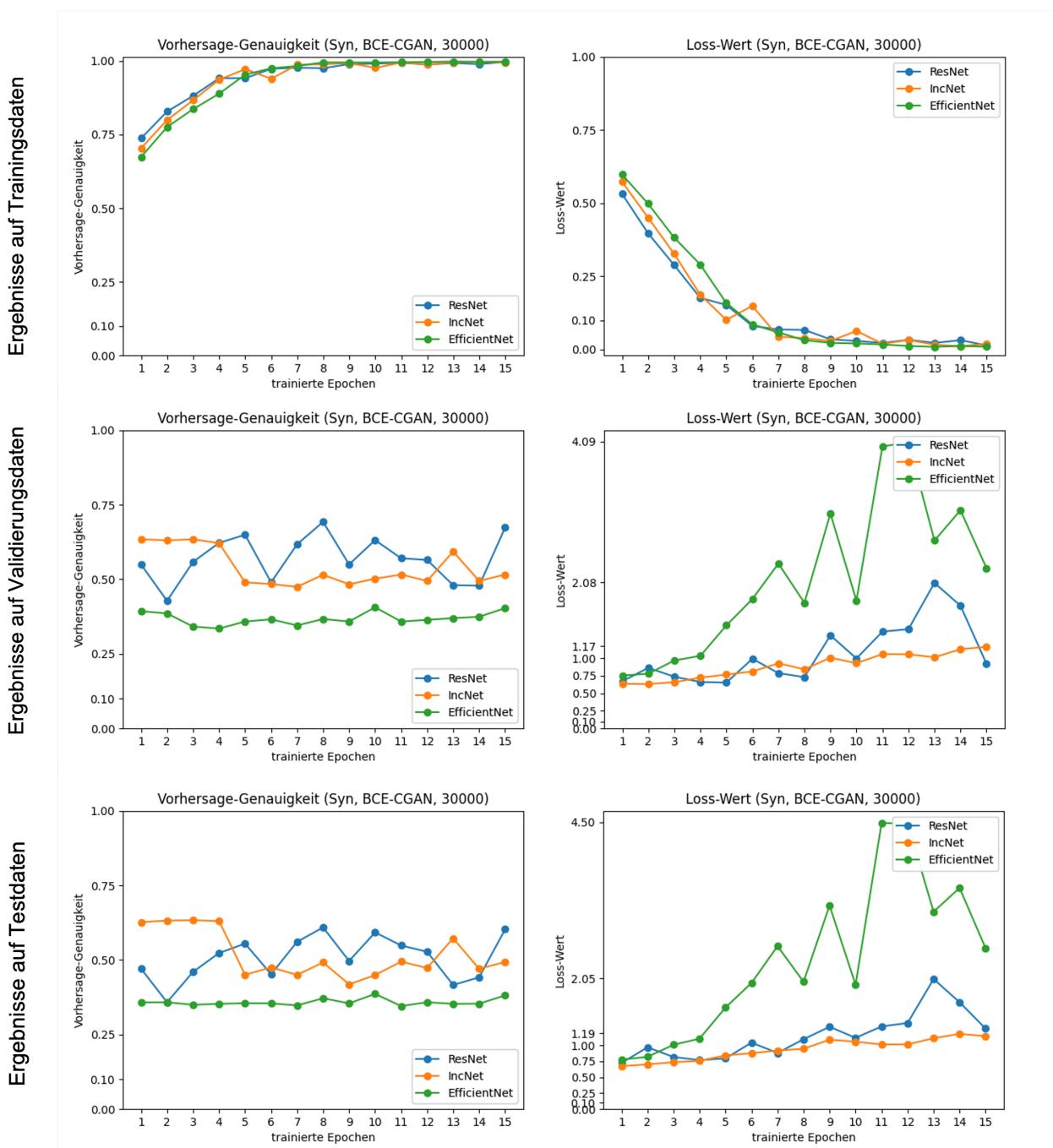


Abbildung A.12.: Testergebnisse eines auf 30.000 mit einem BCE-CGAN generierten synthetischen Daten trainierten Klassifizierungs-KNN:  
Es werden die Vorhersage-Genauigkeit (linke Graphen) und der durchschnittliche Loss-Wert (rechte Graphen) für alle Klassifizierungs-KNN-Konzepte auf den verwendeten Trainingsdaten (oben) sowie den echten Validierungs- (mittig) und Testdaten (unten) abgebildet.

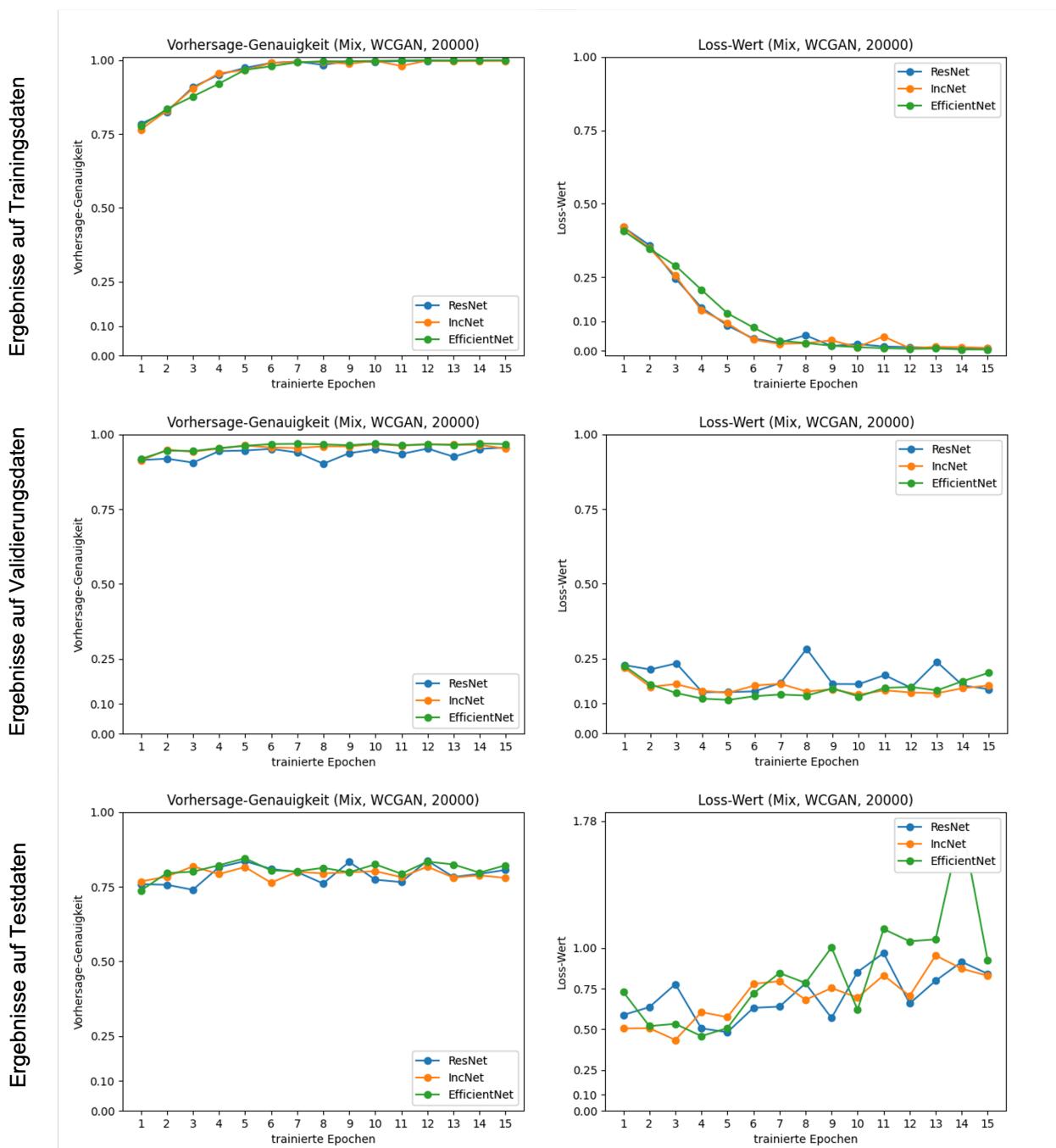


Abbildung A.13.: Testergebnisse eines auf 10.000 mit einem Wasserstein-CGAN generierten synthetischen und 10.000 echten Daten trainierten Klassifizierungs-KNN:

Es werden die Vorhersage-Genauigkeit (linke Graphen) und der durchschnittliche Loss-Wert (rechte Graphen) für alle Klassifizierungs-KNN-Konzepte auf den verwendeten Trainingsdaten (oben) sowie den echten Validierungs- (mittig) und Testdaten (unten) abgebildet.

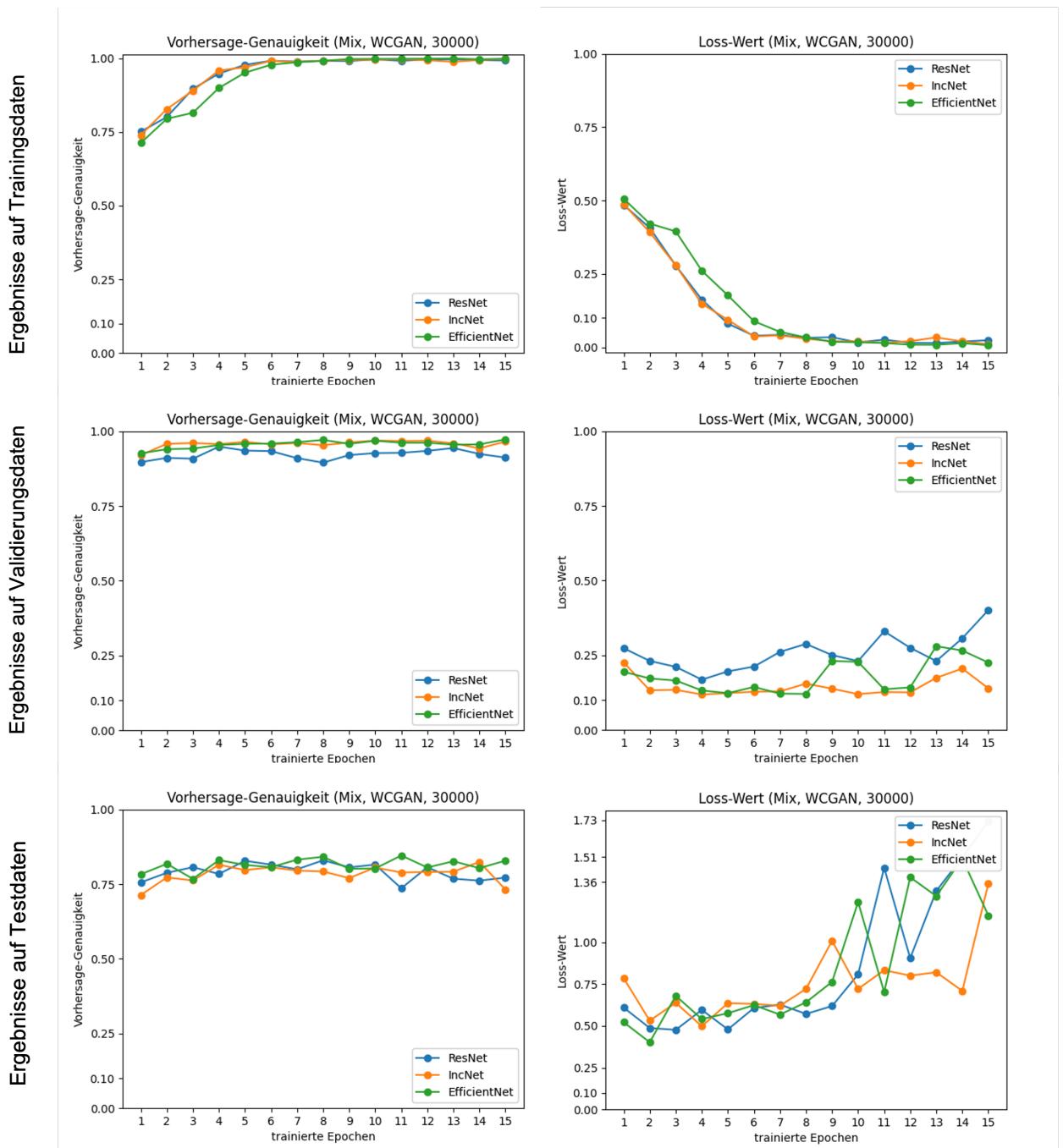


Abbildung A.14.: Testergebnisse eines auf 20.000 mit einem Wasserstein-CGAN generierten synthetischen und 10.000 echten Daten trainierten Klassifizierungs-KNN:

Es werden die Vorhersage-Genauigkeit (linke Graphen) und der durchschnittliche Loss-Wert (rechte Graphen) für alle Klassifizierungs-KNN-Konzepte auf den verwendeten Trainingsdaten (oben) sowie den echten Validierungs- (mittig) und Testdaten (unten) abgebildet.

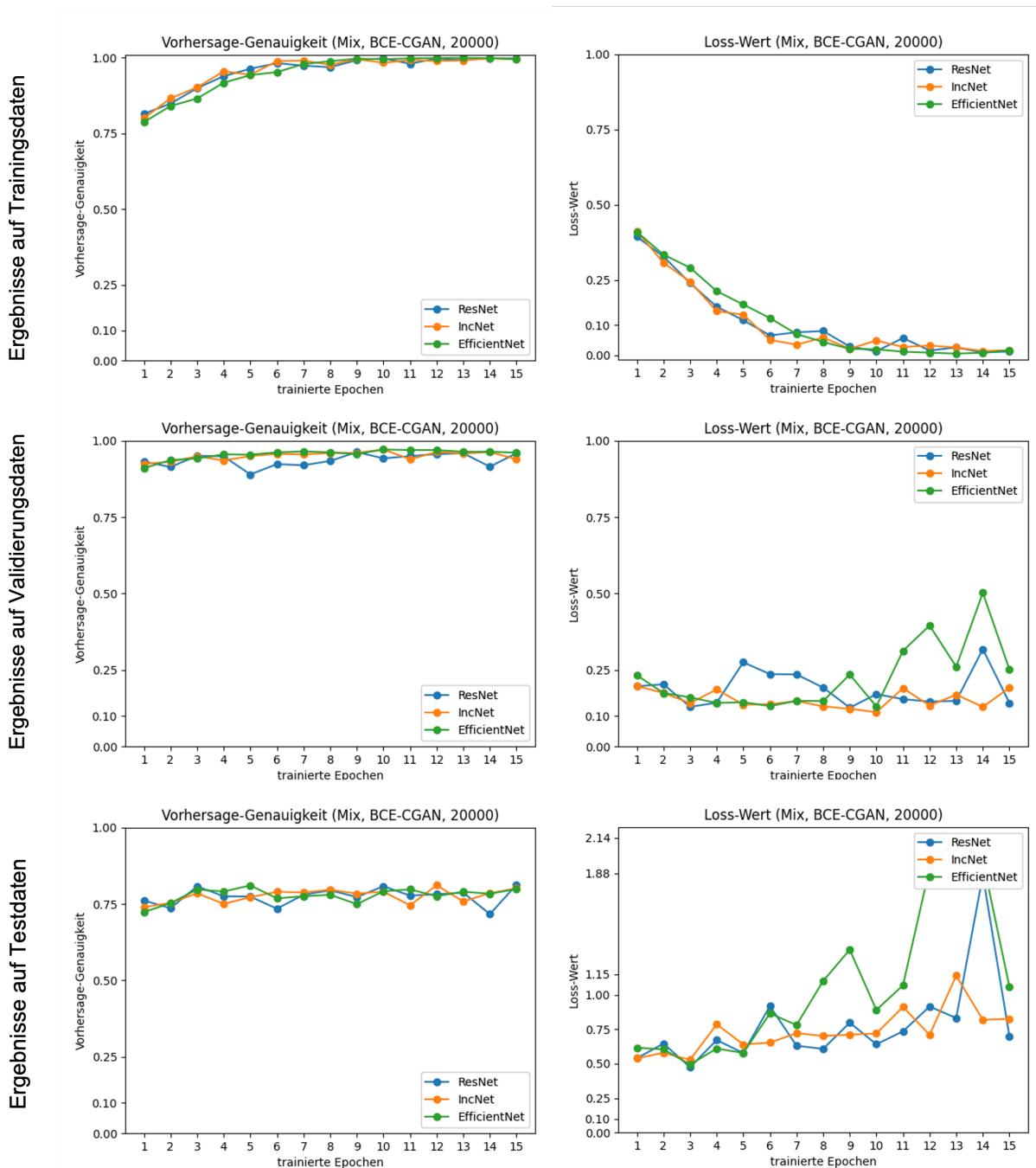


Abbildung A.15.: Testergebnisse eines auf 10.000 mit einem BCE-CGAN generierten synthetischen und 10.000 echten Daten trainierten Klassifizierungs-KNN:

Es werden die Vorhersage-Genauigkeit (linke Graphen) und der durchschnittliche Loss-Wert (rechte Graphen) für alle Klassifizierungs-KNN-Konzepte auf den verwendeten Trainingsdaten (oben) sowie den echten Validierungs- (mittig) und Testdaten (unten) abgebildet.

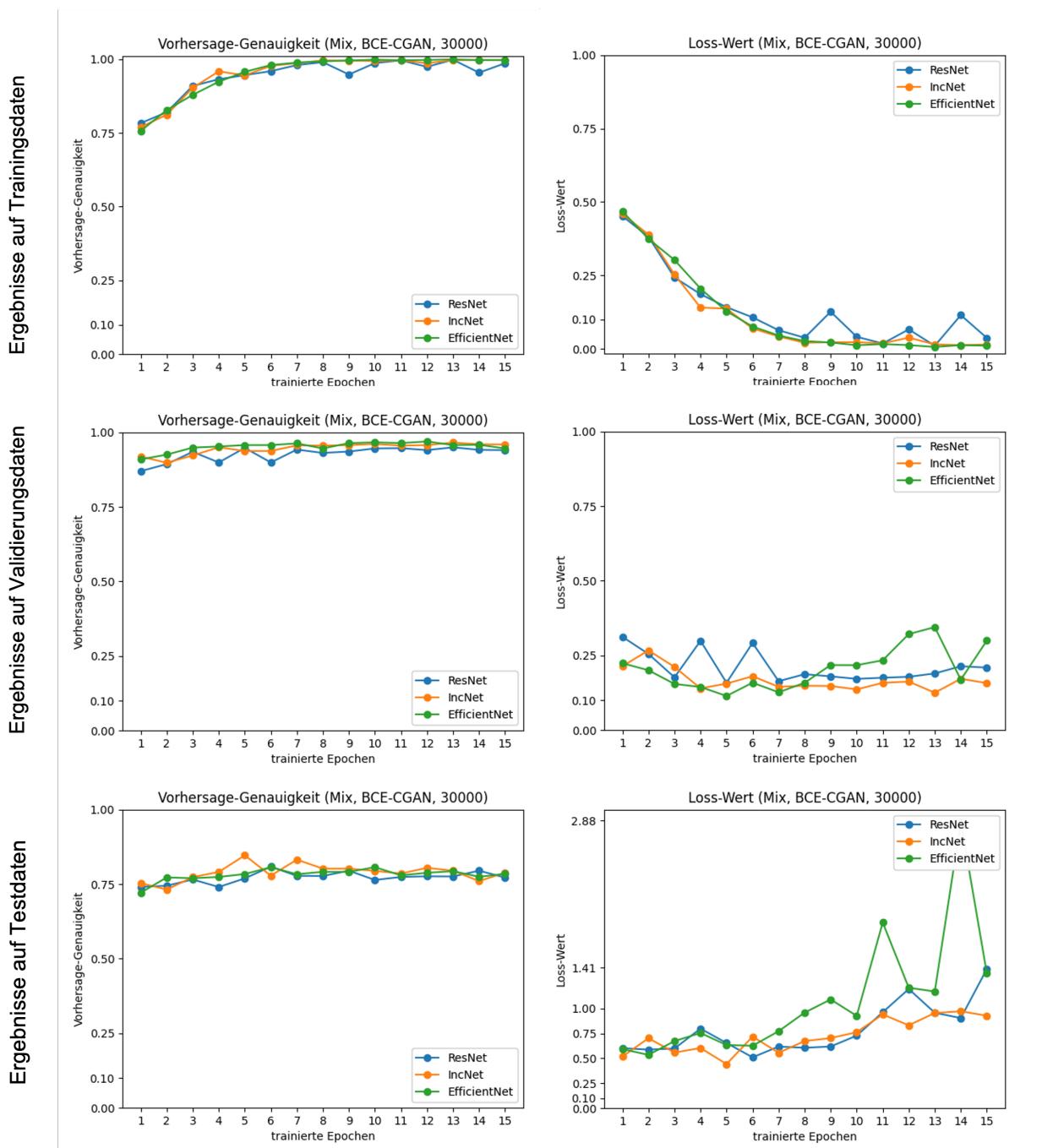


Abbildung A.16.: Testergebnisse eines auf 20.000 mit einem BCE-CGAN generierten synthetischen und 10.000 echten Daten trainierten Klassifizierungs-KNN:

Es werden die Vorhersage-Genauigkeit (linke Graphen) und der durchschnittliche Loss-Wert (rechte Graphen) für alle Klassifizierungs-KNN-Konzepte auf den verwendeten Trainingsdaten (oben) sowie den echten Validierungs- (mittig) und Testdaten (unten) abgebildet.