Kantonsschule Im Lee, Winterthur, Herbstsemester 2019/2020

Maturitätsarbeit

Fotorealistische Bilder mit Hilfe maschinellen Lernens generieren

-

Kennen Sie diese Personen? Kaum!



***Autor*:** **Felix Schnitzler (4fMn)**

***Betreuer*:** **Michael Anderegg**

6. Januar 2020

Inhaltsverzeichnis

1 Einleitung

Im Rahmen dieser Maturitätsarbeit habe ich ein sogenanntes *Generative Adversarial Networks* geschrieben, welches gelernt hat, möglichst fotorealistische Bilder von Gesichtern zu generieren. Dabei handelt es sich nicht um Bilder mit Gesichtern von realen Personen, sondern um Bilder, die aufgrund des vom Algorithmus gelernten Wissens über das allgemeine Aussehen eines Gesichtes generiert wurden. Das Interessante an einem selbstlernenden Algorithmus ist, dass ihm kein Lösungsweg für die Aufgabe einprogrammiert wird, sondern dass dieser selbst lernt und den Lösungsweg findet.

Ich interessiere mich schon seit einiger Zeit für die künstliche Intelligenz und wie selbstlernende Algorithmen in Form von künstlichen neuronalen Netzwerken funktionieren und programmiert werden. Um ein künstliches neuronales Netzwerk zu definieren und zum Trainieren zu bringen, braucht man ein wenig Programmiererfahrung, kann aber durch das Internet schnell einen ausführbaren Programmcode finden. Doch wenn man ein Netzwerk mit eigenen Bildern trainieren möchte, trainiert das Netzwerk unter Umständen nicht richtig. Dafür braucht man ein Verständnis künstlicher neuronaler Netzwerke. Dieses zu erlangen, kann sich als schwierig herausstellen. In dieser Arbeit versuche ich, die Funktionsweise eines *Generative Adversarial Networks erläutern*, sowie eine allgemeine, nicht zu komplizierte Einführung zu künstlichen neuronalen Netzwerken zu geben.

1.1 Anmerkungen

Die Mathematik hinter *Generative Adversarial Networks* ist sehr komplex. Allein die Funktionsweise und die Optimierung eines gewöhnlichen künstlichen neuronalen Netzwerks zu begreifen, erfordert dies das Verständnis von Themen wie Matrizenrechnen, mathematische Optimierung, mehrdimensionaler Analysis. Da *Generative Adversarial Networks* aus mehreren künstlichen neuronalen Netzwerken bestehen, die miteinander interagieren, werden für das Verständnis zusätzlich die Spieltheorie und die Wahrscheinlichkeitsrechnung gebraucht. Für die Implementierung solch eines Netzwerks sind ausserdem noch Programmierkenntnisse nötig. Diese Themen in einer Maturitätsarbeit zu erläutern, damit jeder Aspekt eines *Generative Adversarial Networks* und die Implementierung verstanden wird, ist schlicht unmöglich. Darum habe ich versucht, die Grundidee der Funktionsweise von künstlichen neuronalen Netzwerken und ihr Miteinanderagieren mit so wenig Mathematik wie möglich zu erklären.

2 Maschinelles Lernen

Wenn über Themen wie maschinelles Lernen, selbstlernende Algorithmen und künstliche Intelligenz gesprochen wird, hat man oft das Bild von sprechenden Robotern und Computerprogrammen vor Augen, die ein Selbstbewusstsein erlangt haben. Dafür wird wohl vor allem Hollywood mit seinen Science-Fiction Filmen verantwortlich sein. Dass Computer sich ihrer selbst bewusst werden können, ist und wird für eine lange Weile auch Fiktion bleiben. Es kommt immer mehr die Frage auf, ob Intelligenz überhaupt mit Bewusstsein gekoppelt sein muss, oder ob es auch nicht-bewusste Intelligenz geben kann. Da es keine allgemeine Definition der Intelligenz und des Bewusstseins gibt, lässt sich diese Frage nicht eindeutig beantworten [1, S. 420].

Als maschinelles Lernen bezeichnet man die Automatisierung des Lernens. Normalerweise werden das Wissen und die Fertigkeiten, die Maschinen haben, von Menschen entwickelt und einprogrammiert. Doch durch das maschinelle Lernen können Maschinen Wissen und Fertigkeiten selbst erlernen. Dies erweist sich als sehr nützlich, wenn es um eine Aufgabe geht, deren Lösungsweg kompliziert oder nicht bekannt ist. Wie zum Beispiel unterscheidet man Katzen von Hunden oder erkennt Emotionen in Gesichtern? Solche Aufgaben sind für uns Menschen kinderleicht, doch wie würden Sie einen Lösungsweg definieren, mit dem eine Maschine programmiert werden kann? Solch einen Algorithmus zu finden, der auch nur ansatzweise mit einem Menschen mithalten kann, ist keineswegs einfach. Bei noch komplizierteren Aufgaben wie der Vorhersage von Aktienentwicklungen oder die frühzeitige Krebserkennung gibt es zu viele Variablen und deren Zusammenhänge, um ohne jahrelange Forschung und Studien einen verlässlichen, mathematisch formulierbaren Lösungsweg zu finden.

2.1 Wie ein Algorithmus lernen kann

Um herauszufinden, wie ein Algorithmus Wissen generieren kann, muss untersucht werden, wie die Menschen das Wissen zuvor erlangt haben. Komplizierte Aufgaben mit vielen Variablen versuchen Wissenschaftler zu lösen, indem sie durch Statistiken und Studien die Zusammenhänge der Variablen finden. Im Allgemeinen ist für das Erlernen neuen Wissens schon vorhandenes Wissen und Erfahrung in Form von Daten notwendig.

Ein selbstlernender Algorithmus muss also die Zusammenhänge der Variablen aus diesen Daten selbst erlernen.

Ein Ansatz wäre, dass der Algorithmus die Daten auswendig lernt. Dadurch braucht er die Zusammenhänge nicht zu finden. Doch mit dieser Methode können über neue, unbekannte Daten keine Aussagen gemacht werden [2]. Wenn also ein Algorithmus Millionen von Bildern von Katzen und Hunden mit deren Bezeichnung auswendig lernt, kann der Algorithmus perfekt diese Bilder unterscheiden. Doch sobald er eine Aussage über ein neues Bild machen soll, das sich nicht in seinem Speicher befindet, ist er hilflos. Um auch neue Bilder unterscheiden zu können, muss der Algorithmus also Muster und Zusammenhänge zwischen allen Hunden und allen Katzen finden und ein abstraktes, verallgemeinertes Konzept von Hunden und Katzen erstellen. Dieses abstrakte Bild entsteht in einem mathematischen Modell, das wie eine Funktion die Daten mit den Ergebnissen verknüpft. Im Fall des Beispiels mit der Unterscheidung von Hunden und Katzen könnte der Eingabewert der Funktion die Pixelwerte eines Bildes sein. Der Funktionswert könnte dann ausgeben, mit welcher Wahrscheinlichkeit es sich um eine Katze oder einen Hund handelt.

Maschinelles Lernen ist aber sehr effektiv. Firmen und Internetseiten analysieren auf diese Weise das Verhalten der Kunden und Nutzer und können personalisierte Werbung versenden. Eine Supermarktkette in den USA konnte zum Beispiel die Schwangerschaft einer Teenagerin aufgrund ihres Kaufverhaltens noch vor den Eltern entdecken. Als an die Tochter adressierte Coupons für Babykleidung in der Post lagen, beschwerten sich die Eltern beim Filialleiter des Supermarkts. Erst einige Tage später erfuhren die Elterrn von der Schwangerschaft der Tochter und zogen die Beschwerde zurück [3, S. 243f.].

3 Künstliche neuronale Netzwerke

Wenn es um das Erstellen eines Modells geht, mit dem man Zusammenhänge zwischen verschiedenen Daten finden und speichern kann, dient unser Gehirn wieder als Vorbild. Das Gehirn ist ein Netzwerk aus unzähligen Neuronen, die miteinander kommunizieren. Es kann schon mit einigen Jahren Erfahrung lernen, Katzen von Hunden zu unterscheiden und Gesichter auseinanderzuhalten. Wie sich herausstellt, sind auch virtuelle mathematische Modelle eines biologischen neuronalen Netzwerks in der Lage, solche Aufgaben zu bewältigen. In diesem Kapitel werden der Aufbau und die Optimierung eines künstlichen neuronalen Netzwerks beschrieben.

3.1 Aufbau künstlicher neuronaler Netzwerke

Das neuronale Netzwerk in einem Gehirn ist sehr kompliziert aufgebaut, und dessen Neuronen sind mit vielen anderen Neuronen verbunden. Beim Versuch, ein biologisches neuronales Netzwerk mathematisch zu formulieren, wurde etwas Ordnung geschaffen. Das Netzwerk wurde in verschiedenen Schichten aufgeteilt, die miteinander verbunden sind. Beginnend mit einer Eingabeschicht hört das Netzwerk mit einer Ausgabeschicht auf. Alle Schichten dazwischen werden verborgene Schichten genannt [4, S. 4]. Auf diese Weise können Daten in das Netzwerk eingespeist werden und fliessen durch das Netzwerk bis zur Ausgabeschicht. Um auf das Beispiel der Unterscheidung von Katzen- und Hundebildern zurückzukommen, wird ein Bild in die Eingabeschicht des Netzwerks eingespeist und fliesst bis zur Ausgabeschicht durch, die eine Zahl ausgibt. Je nachdem ob diese Zahl positiv oder negativ ist, handelt es sich um eine Katze oder einen Hund.

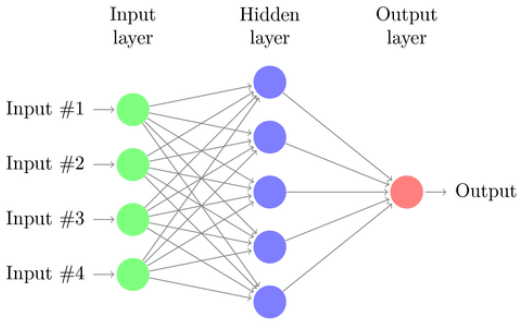

Abb. 3.1: Ein künstliches neuronales Netzwerk mit drei Schichten. Die Eingabeschicht (Input Layer), die Ausgabeschicht (Output Layer) und eine Schicht dazwischen (Hidden Layer). Die Neuronen einer Schicht sind durch Gewichte mit den Neuronen der nächsten Schicht verbunden.

Die Schichten des Netzwerks bestehen aus künstlichen Neuronen. Diese stellen eine mathematische Vereinfachung eines biologischen Neurons dar. Ein künstliches Neuron besitzt einen Wert. Der Wert, auch Netzeingabe genannt, ergibt sich aus der gewichteten Summe der Eingabe des Neurons. Die Eingabe X steht für alle Aktivierungen der Neuronen in der vorangehenden Schicht, die mit diesem Neuron verbunden sind. Da die Signale

der Neuronen unterschiedlich viel Einfluss auf das nächste Neuron haben können, werden durch Multiplikation der Gewichte *w* die Aktivierung beziehungsweise die Eingabe der einzelnen Neuronen verstärkt oder gehemmt. Die gewichtete Summe dieser Eingaben werden durch die Übertragungsfunktion zur Netzeingabe bestimmt. Im Normalfall bildet die Übertragungsfunktion die Summe aller gewichteten Eingaben. Der Netzeingabe wird der Schwellwert θ hinzuaddiert und durch die Aktivierungsfunktion φ aktiviert. Diese Aktivierung wird als Eingabe der Neuronen in der nächsten Schicht verwendet [4, S. 2ff.] [5].

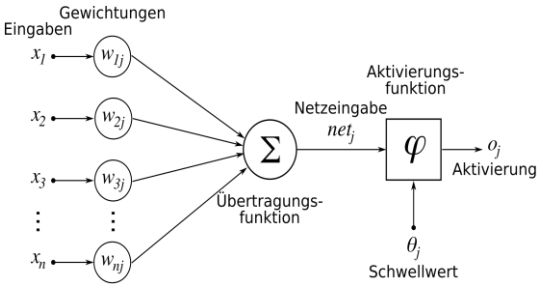

Abb. 3.2: Darstellung eines künstlichen Neurons mit seinen Elementen. Die Aktivierungen der verbundenen Neuronen (x) werden mit den Gewichten (w) multipliziert und im Neuron durch die Übertragungsfunktion zu einem Wert addiert. Die Aktivierung des Neurons wird durch die Aktivierungsfunktion (φ) und den Schwellenwert (θ) bestimmt.

Da eine Schicht mehrere künstliche Neuronen besitzt, kann die ganze Schicht als Operationen mit Matrizen dargestellt werden. Dies hat den Vorteil, dass auch mehrere Daten auf einmal durch das Netzwerk fliessen können. Für das Katze-Hund-Beispiel würde dies heissen, dass mehrere Bilder auf einmal durch das Netzwerk verarbeitet werden können und Vorhersagen erstellt werden können.

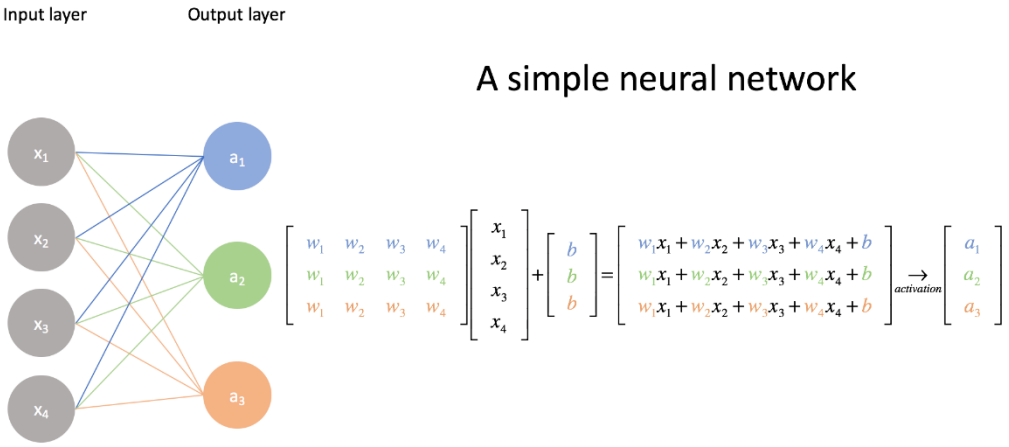

Abb. 3.3: Ein künstliches neuronales Netzwerk als Matrix-Multiplikation. Die Eingaben (x) werden mit den Gewichten (w) multipliziert und der Schwellwert (b) wird addiert. Die Ausgaben-Matrix Aktivierungen (a) wird durch die Aktivierung der Netzeingabe-Matrix erhalten.

Für die Anzahl an Neuronen, die ein Netzwerk haben soll, damit es die Daten verallgemeinern kann, gibt es keine wirkliche Regel. Ist die Anzahl zu klein, kann das Netzwerk nicht genügend Muster in den Gewichten speichern und es lernt zu wenig. Man spricht von *Underfitting*, auf Deutsch Unteranpassung. Besitzt das Netzwerk zu viele Neuronen, lernt das Netzwerk u.U. zu viele Regeln, d.h., es lernt jeden einzelnen Fall als Regel und speichert so die Daten in den Gewichten und lernt sie auswendig. Diesen Fall nennt man *Overfitting*, deutsch Überanpassung. Es gilt, ein Mittelmass zwischen dem *Under-* und *Overfitting* zu finden [4, S. 5].

3.2 Training eines künstlichen neuronalen Netzwerks

Die Gewichte des künstlichen neuronalen Netzwerks sind die einzigen veränderlichen Werte oder Parameter. Diese Gewichte bestimmen ausserdem, wie das Netzwerk bei einer Eingabe die Ausgabe berechnet. Doch wie stellt man die richtigen Gewichte ein, sodass das Netzwerk korrekte Vorhersagen macht?

Damit das künstliche neuronale Netzwerk lernen kann, braucht man Trainingsdaten. Angenommen wir haben die Beispieldaten x und y. Das Netzwerk soll den Zusammenhang dieser Daten lernen. Wenn das Netzwerk als Eingabe x erhält, soll es den Wert y in der Ausgabeschicht ausgeben. Da von x auf y geschlossen werden soll, nennt man x das *Feature* und den gewünschten Wert y bezeichnet man als *Target* oder *Label*. Die Ausgabe des Netzwerks wird als *Vorhersage* bezeichnet und erhält das Symbol $\tilde{y}$.

Weil die Gewichte anfangs zufällig eingestellt sind, berechnet das Netzwerk auch die Vorhersage als einen zufälligen Wert. Durch Vergleichen der Vorhersage $\tilde{y}$ des Netzwerks und des wahren Labels y können die Gewichte so verändert werden, dass das Netzwerk den erwünschten Wert ausgibt. Beim Katze-Hund-Beispiel könnte bei einem untrainierten Netzwerk die Vorhersage bei einem Katzenbild eine negative Zahl sein. Diese negative Zahl steht dafür, dass das Netzwerk meint, es handle sich um einen Hund. Danach passt man die Gewichte so an, dass die Vorhersage zu einer positiven Zahl wird, was für die Katze steht.

Doch wie stellt man die Gewichte ein, damit die Vorhersage mit dem gewünschten Label übereinstimmt, und zwar nicht für ein, sondern für die meisten Datenpaare?

3.2.1 Fehlerfunktion

Für die Optimierung eines künstlichen neuronalen Netzwerks muss bestimmt werden können, wie gut die Vorhersagen des Netzwerks mit den Labels übereinstimmen, wie gross also der Fehler ist, den das Netzwerk beim Vorhersagen macht. Für einfache Netzwerke werden normalerweise die Differenzen zwischen der Vorhersage und des Labels berechnet. Damit es keine negativen Werte gibt, wird die Differenz quadriert. Der durchschnittliche Fehler für mehrere Daten ergibt dann die Fehlerfunktion. Diese Art von Fehlerfunktion wird auch *mittlere quadratische Abweichung* genannt.

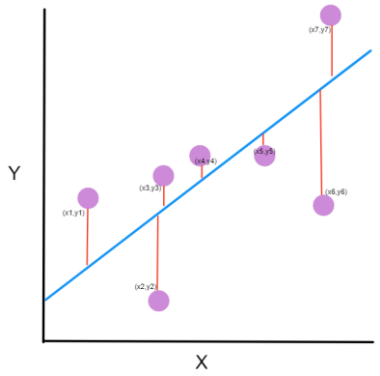

Abb. 3.4: Jeder X-Wert hat einen gewünschten Y-Wert (Gerade). Die Voraussagen des Netzwerks (Punkte) weichen von den gewünschten Ergebnissen ab. Um die Fehler zu berechnen, werden die Differenzen der Punkte quadriert und der Durchschnitt davon gebildet.

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$$

Abb. 3.4: Formel für die mittlere quadratische Abweichung (mean squared error - MSE). Es wird der Durchschnitt der Differenzen zwischen den wahren Labels (y) und den Vorhersagen (ỹ) gebildet.

3.2.2 Optimierungsverfahren

Je kleiner der Fehler der Fehlerfunktion ist, desto besser kann das Netzwerk die richtigen Labels vorhersagen. Anders ausgedrückt muss bei der Optimierung eines künstlichen neuronalen Netzwerks die Fehlerfunktion minimiert werden. Damit der Funktionswert der Fehlerfunktion sich ändert, müssen deren Parameter geändert werden. Wie man in Abbildung 3.5 sieht, hängt die Fehlerfunktion vom Label y und der Vorhersage $\tilde{y}$ ab. y ist unveränderlich, also muss sich $\tilde{y}$ ändern. Die Vorhersage des Netzwerks hängt, wie wir oben gesehen haben, nur von der unveränderlichen Eingabe und den veränderlichen Gewichten ab. Somit sind die eigentlichen Parameter der Fehlerfunktion die Gewichte.

Für die Optimierung des Netzwerks beziehungsweise Minimierung der Fehlerfunktion gibt es verschiedene Verfahren. Da diese sehr kompliziert sein können, aber im Grunde alle dasselbe bewirken, werde ich nur auf einen grundlegenden Optimierungsalgorithmus eingehen. Es handelt sich dabei um das *Gradientenverfahren*. Dieses Verfahren verwendet die erste Ableitung der Fehlerfunktion. Da die Funktion von mehreren Parametern abhängt, ist sie mehrdimensional, und anstatt der Ableitung berechnet man den Gradienten, der eine Verallgemeinerung dieser Ableitung ist. Aus dem Gradienten der Fehlerfunktion kann die Art und Weise geschlossen werden, wie die Parameter des Netzwerks geändert werden müssen, d.h. ob die Parameter etwas grösser oder kleiner sein müssen, um die Fehlerfunktion zu minimieren. Das Gradientenverfahren ist ein iteratives Verfahren, bei dem die Anpassung der Parameter durch die Berechnung des Gradienten mehrmals angewendet werden muss [6] [7].

3.3 Schichten eines künstlichen neuronalen Netzwerks

Die Schichten in einem künstlichen neuronalen Netzwerk können sehr unterschiedlich aussehen. Je nachdem, bei welcher Art von Daten das Netzwerk die Zusammenhänge zwischen ihnen erlernen soll, sind nicht alle Arten von Schichten gleich effektiv. In diesem Unterkapitel gehe ich auf zwei Schichten ein, die für das Verständnis des in Kapitel 4 behandelten *Generative Adversarial Networks* wichtig sind, welches das Hauptthema meiner Arbeit ist. Ausserdem werde ich die in meinem GAN verwendete Aktivierungsfunktionen *Leaky ReLU* und *Sigmoid* sowie eine Methode zu Stabilisierung des Trainings erläutern. Ich werde allerdings nicht allzu ausführlich die Funktionsweise erläutern, da der mathematische Hintergrund sehr komplex ist.

3.3.1 Fully Connected Layer

Ein *Fully Connected Layer*, auf Deutsch *vollständig verbundene Schicht* oder auch *Dense Layer*, zu Deutsch *dichte Schicht* ist die einfachste Schicht. Sie ist vollständig mit der vorhergehenden Schicht verbunden. Jedes Neuron eines *Fully Connected Layers* ist mit allen Neuronen in der Schicht davor verbunden.

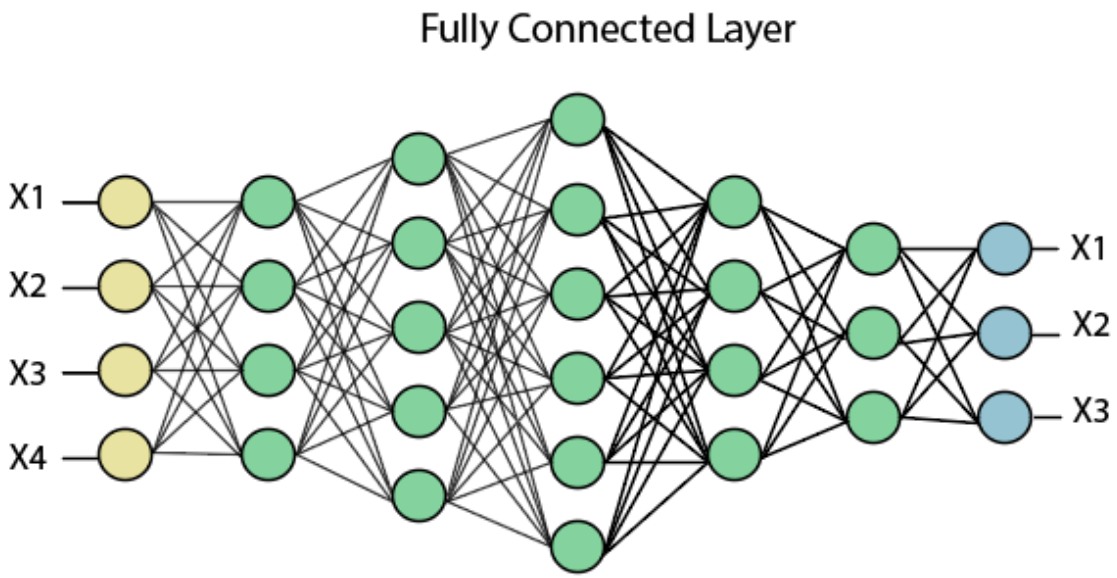

Abb. 3.5: Fully Connected Layer - jedes Neuron einer Schicht ist mit allen Neuronen der vorhergehenden Schicht verbunden.

3.3.2 Convolutional Layer

Ein *Convolutional Layer* ist eine Schicht, die verwendet wird, wenn das Netzwerk von Bildern lernen soll, denn dann eignen sich vollständig verbundene Schichten nicht mehr sehr gut. Die Netzwerke können bei Bildern sehr schnell sehr viele Neuronen und Gewichte haben und es tritt Overfitting, auf bei dem das Netzwerk nicht allgemeine Regeln findet, sondern die Trainingsdaten auswendig lernt [4, S. 5].

Ein *Convolutional Layer* kommt hingegen mit weit weniger Parametern aus. Bei dieser Art von Schicht lernt das Netzwerk, die wichtigen Eigenschaften in einem Bild herauszufiltern und die unwichtigen wegzulassen. Dadurch wird weniger Rechenleistung gebraucht. Zum Beispiel kann ein *Convolutional Layer* Ecken aus dem Bild filtern. Nach mehreren *Convolutional Layern* können auch abstraktere Eigenschaften wie Nasen, Ohren, Buchstaben oder ähnliches herausgefiltert werden. Die Ausgabe eines *Convolutional Layers* ist ein neues Bild, das an den Stellen, an denen der Filter gepasst hat, hell und an den anderen Stellen dunkel ist. Dieser Filter kann als Matrix dargestellt werden und mit einer mathematischen Operation, der *Konvolution*, auf das Bild angewendet werden. Welche Werte die Filter-Matrix hat, wird das Netzwerk lernen. Die Filter können als Gewichte der Netzwerkschicht angesehen werden. Das Netzwerk versucht, die optimalen Filter zu finden, um die Fehlerfunktion zu minimieren. Dies kann durch dieselben Optimierungsverfahren geschehen [8].

Abb. 3.6: Durch eine Konvolution wurden die Kanten eines Hauses erkannt.

3.3.3 Rectified Linear Unit (ReLU)

Die *Rectified Linear Unit (ReLU)* ist eine Aktivierungsfunktion, die in künstlichen neuronalen Netzwerken verwendet wird. Sie ist seit 2017 die beliebteste Aktivierungsfunktion [9]. Diese wird auch für ein stabiles Training von *Generative Adversarial Networks* mit *Convolutional Layern* verwendet [10]. Der Vorteil, den sie mit sich bringt, ist, dass der zu berechnende Gradient nicht zu klein wird und nicht an Information verliert, wodurch nicht die optimalen Parameter des Netzwerks gefunden werden können. Die Funktion einer *Rectified Linear Unit* lautet

$$f(x) = max(0.01x, x)$$

und gibt den eingabewert wieder aus, falls dieser positiv ist und dividiert ihn durch 100, falls er negativ ist [11].

3.3.4 Sigmoidfunktion

Bei der *Sigmoidfunktion* handelt es sich um eine weitere Aktivierungsfunktion. Sie wird meist in der Ausgabeschicht genutzt, wenn die Ausgabe zwischen Null und Eins sein soll. Solch eine Ausgabe kann dann als Wahrscheinlichkeit interpretiert werden. Zum Beispiel ist es bei der Unterscheidung von Katzen und Hunden von Vorteil, die Vorhersage des Netzwerks als Wahrscheinlichkeit zu betrachten. Die Vorhersage kann als Wahrscheinlichkeit verstanden werden, wie sicher sich das Netzwerk bei der Unterscheidung ist. Ist sich das Netzwerk sicher, dass es sich um eine Katze in einem Bild handelt, dann ist die Vorhersage nahe bei Eins. Ist sich das Netzwerk sicher, dass sich nicht um eine Katze, sondern um einen Hund handelt, ist die Vorhersage nahe bei Null. Ist sich das Netzwerk jedoch nicht sicher, ob das Bild eine Katze oder einen Hund darstellt, wird sich die Vorhersage bei 0.5 befinden.

Um die Ausgabe des Netzwerks auf Zahlen zwischen Null und Eins zu beschränken, wird auf die normale Ausgabe die Sigmoid-Aktivierungsfunktion angewendet. Die Grenzwerte der Sigmoidfunktion sind Null und Eins:

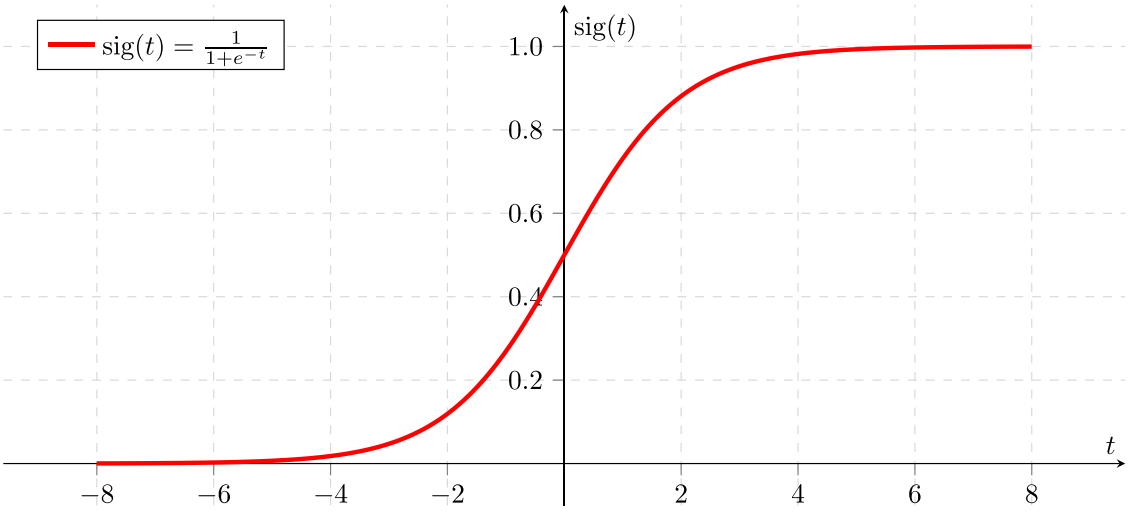

Abb. 3.7: Der Grenzwert der Sigmoidfunktion sind Null und Eins. Eine stark negative Ausgabe wird auf Null und eine stark positive Ausgabe auf Eins gesetzt.

3.3.5 Batch Normalisierung

Um das Training eines künstlichen neuronalen Netzwerks weiter zu stabilisieren, werden die Werte nach jeder Schicht normalisiert. Es wird dabei jeweils das sogenannte *Batch* normalisiert. Ein Batch kann etwa mit Stapel oder Päckchen übersetzt werden und beinhaltet alle Trainingsdaten, die auf einmal durch das Netzwerk fliessen und für einen Optimierungsschritt verwendet werden. Vereinfacht erklärt, wird für die Normalisierung der Durchschnittswert und die Standartabweichung des Batch berechnet. Der Durchschnitt wird von jedem Wert im Batch subtrahiert und durch die Standardabweichung dividiert. Die Batch-Normalisierung besitzt zwei trainierbare Parameter [12].

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Abb. 3.8: Algorithmus für die Batch-Normalisierung. μ ist der Durchschnitt und σ die Standartabweichung in einem Batch.

4 Generative Adversarial Networks (GAN)

Anstatt Bilder von Haustieren zu erkennen, können künstliche neuronale Netzwerke auch neue Bilder von Haustieren generieren. Diese Netzwerke lernen zum Beispiel, wie eine Katze aussieht, und können neue Bilder generieren, welche die Kriterien für ihr Konzept einer Katze erfüllen. Mit diesen *Generative Adversarial Networks* (kurz GAN) können also neue Daten generiert werden, die den Trainingsdaten ähnlich sind.

4.1 Funktionsweise eines GAN

Die Idee hinter einem generativen Netzwerk ist, dass man die Daten der Ein- und Ausgabeschicht vertauscht. Somit ist die Ausgabe des Netzwerks ein Bild und die Eingabe die dazugehörige Bezeichnung. Das Netzwerk wird sozusagen umgekehrt und die Labels werden mit den Features getauscht. Anstatt dass das Netzwerk die Ausgabe hervorsagt, wird davon gesprochen, dass es die Ausgabe *generiert*. Dies ergibt in diesem Kontext mehr Sinn, als von einer Vorhersage des Netzwerks zu sprechen.

Aus der Umkehrung des Netzwerks entsteht aber ein Problem. Bei der Aufgabe der Bilderkennung gibt es viele verschiedene Features, die dasselbe Label haben. Es gibt viele unterschiedliche Bilder, die alle eine Katze abbilden. Doch kehrt man das Netzwerk um, sollen aus ein und demselben Label unterschiedliche Bilder vorhergesagt werden. Das Netzwerk kann lernen, aus einem Label immer das gleiche Bild zu generieren. Aber bei gleichbleibender Eingabe wird die Ausgabe des Netzwerks nicht jedes Mal ein anderes Bild generieren. Dieses Problem wird dadurch umgangen, dass die Eingabe sich auch jedes Mal ändert. Die Eingabe wird durch einen Zufallsgenerator bestimmt.

2014 veröffentlichen Ian Goodfellow et al. ein Paper, in dem vorgeschlagen wurde, dass ein generatives Netzwerk durch einen spieltheoretischen Kampf zweier Netzwerke trainiert werden kann [13]. Daher rührt auch der Name des *Generative Adversarial Network*, der auf Deutsch etwa mit „Erzeugendes gegnerisches Netzwerk“ übersetzt werden kann.

Diesen Kampf kann man sich als Wettbewerb zwischen einem Geldfälscher und einem Polizisten vorstellen. Der Geldfälscher möchte eine Banknote fälschen. Anfangs ist er unerfahren und für den Polizisten ist es leicht, die Fälschung zu erkennen. Da der Polizist dem Geldfälscher aber jedes Mal eine Rückmeldung gibt, was die Fälschung verraten hat, kann der Fälscher sich stets verbessern. Für den Polizisten wird es immer schwieriger,

die Fälschung von einer echten Banknote zu unterscheiden. Nach einigen Versuchen des Geldfälschers wird dieser so gut, dass der Polizist den Unterschied nicht mehr erkennen kann [14].

Die beiden Personen stehen für zwei künstliche neuronale Netzwerke. Das erste Netzwerk generiert wie der Geldfälscher Bilder, und das zweite versucht, wie der Polizist die generierten Bilder von den echten zu unterscheiden. Das generierende Netzwerk wird *Generator* und das unterscheidende Netzwerk wird *Diskriminator* genannt. Der Diskriminator gibt seine Vorhersage als Wahrscheinlichkeit aus, dass es sich um ein echtes Bild handelt. Somit ist die Ausgabe des Diskriminators immer zwischen Null und Eins.

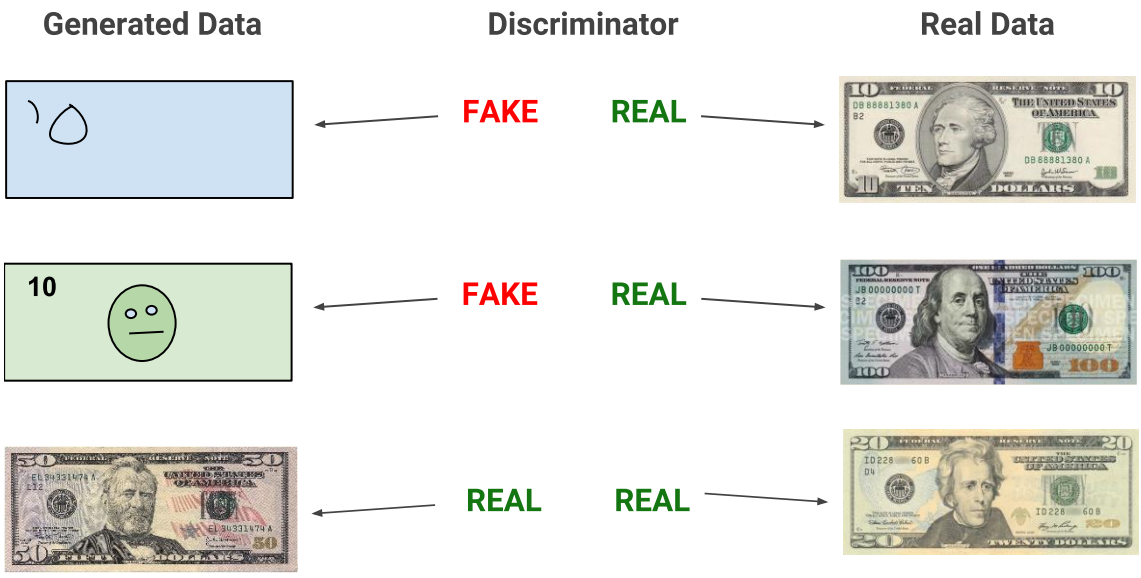

Abb. 4.1: Der Generator wird immer besser darin, eine Banknote zu fälschen, da der Diskriminator ihm eine Rückmeldung gibt. Der Diskriminator vergleicht das generierte Bild nicht immer mit ein und demselben Trainingsbild, da der Generator sonst nur lernt, wie er das eine Bild, aber keine neuen Bilder generiert.

Die Gewichte des Generators werden beim Training so optimiert, dass der Diskriminator eine Vorhersage für ein echtes Bild von den Trainingsdaten macht. Die Gewichte des Diskriminators werden so optimiert, dass er die generierten von den echten Bildern unterscheiden kann. Die Optimierung des Generators und des Diskriminators erfolgt nicht gleichzeitig, sondern abwechselnd nacheinander.

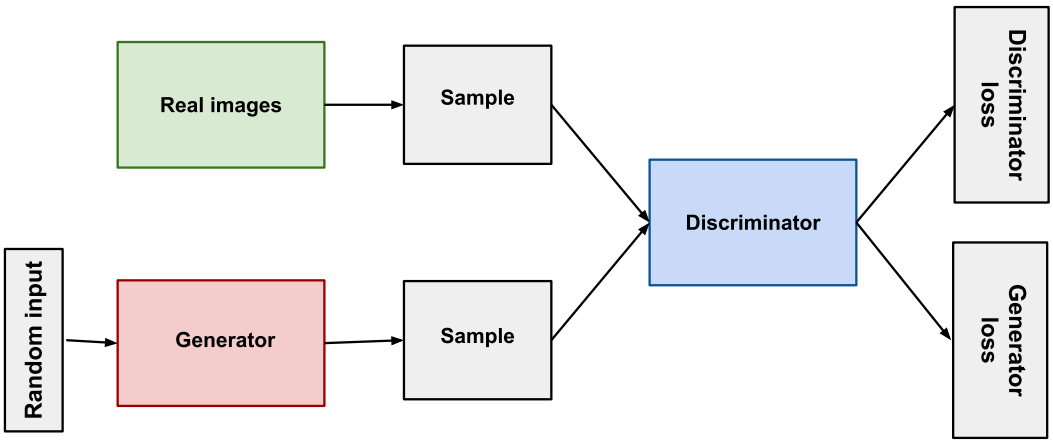

Abb. 4.2: Der Generator erhält zufällige Eingaben und generiert Bilder. Der Diskriminator versucht zu unterscheiden, ob das Bild (Sample) von echten Bildern oder vom Generator stammt. Da beide Netzwerke optimiert werden, wird aus der Vorhersage des Diskriminators der Fehler des Generators (Generator loss) und der Fehler des Diskriminators (Discriminator loss) bestimmt.

Das theoretische Ziel des Trainings ist, dass der Generator und der Diskriminator sich in einem sogenannten *Nash-Gleichgewicht* befinden, d.h., dass beide nur schlechter werden, wenn sie die Parameter ändern. Dieses Ziel ist schwierig zu erreichen, aber theoretisch wäre es möglich.

4.2 Fehlerfunktion eines GAN

Die Ziele des Generators und des Diskriminators sind genau gegenteilig. Der Diskriminator wird optimiert, um seine Fehlerfunktion zu minimieren, da er alle Bilder richtig unterscheiden soll. Der Generator hingegen wird so optimiert, dass die Fehlerfunktion des Diskriminators maximiert wird, weil er den Diskriminator täuschen soll. Die Fehlerfunktion eines *Generative Adversarial Networks* ist somit die Fehlerfunktion des Diskriminators. Sie gibt den Abstand zwischen der Vorhersage des Diskriminators und des wahren Labels an. Anstatt die mittlere quadratische Abweichung zu benutzen (siehe Kapitel 3.2.1), wird die sogenannte *Kreuzentropie* als Fehlerfunktion verwendet. Diese stammt aus der Informationstheorie und ist ein wenig komplizierter als die mittlere quadratische Abweichung. Die Kreuzentropie verständlich zu erklären, würde den Rahmen dieser Maturitätsarbeit sprengen. In diesem Fall aber kann sie als Funktion verstanden werden, welche die Vorhersage des Netzwerks mit den wahren Labels vergleicht und, je besser diese übereinstimmen, desto näher liegt der Funktionswert bei Null. Schlechtere Vorhersagen ergeben einen höheren Wert bei der Kreuzentropie.

Die Fehlerfunktion eines *Generative Adversarial Network* besteht aus zwei Termen. Der erste Term ist der Fehler $E_x[logD(x)]$, den der Diskriminator beim Vorhersagen der Labels echter Bilder von den Trainingsdaten macht. Der zweite Term der Fehlerfunktion bezieht sich auf den Fehler $E_z[log(1 - D(G(z)))]$ beim Vorhersagen der Labels für generierte Bilder des Generators. Diese Schreibweise ist möglicherweise etwas irritierend, doch wird sie verwendet, um die Kreuzentropie anzugeben [15].

$$E_x[log(D(x))] + E_z[log(1 - D(G(z)))]$$

Abb. 4.3: Die Fehlerfunktion eines Generative Adversarial Networks besteht aus zwei Termen. Der erste Term ist der Fehler E_x, den der Diskriminator bei Trainingsdaten macht, und der zweite Term bezieht sich auf den Fehler E_z bei Bildern des Generators.

D und *G* stellen die Diskriminator- und Generator-Netzwerke als Funktion da. *D* gibt die Wahrscheinlichkeit an, mit welcher der Diskriminator das Eingabebild für echt hält. Das *x* steht für die Bilder des Datensatzes und *G(z)* für die vom Generator generierten Bilder. *z* ist die zufällige Eingabe für den Generator *G*. Somit ist *D(x)* die Wahrscheinlichkeit, mit welcher der Diskriminator ein Bild der Trainingsdaten für echt hält. *D(G(z))* ist die Wahrscheinlichkeit, mit welcher der Diskriminator ein generiertes Bild des Generators für echt hält. *1 - D(G(z))* ist somit die Gegenwahrscheinlichkeit, dass der Diskriminator die generierten Bilder nicht für echt, sondern für unecht, also generiert hält.

Der Diskriminator wird so optimiert, dass diese beiden Fehler minimiert werden und die Vorhersagen immer besser stimmen. Der Generator aber wird so optimiert, dass diese Fehler möglichst gross sind. Auf den ersten Term $E_x[logD(x)]$ hat der Generator keinen Einfluss und somit kann durch ihn nur der zweite Fehler $E_z[log(1 - D(G(z)))]$ maximiert werden [14].

5 Programmierung eines GAN

Bevor das *Generative Adversarial Network* trainiert werden kann, müssen die Netzwerke und der Trainingsablauf erst definiert werden. Das GAN beinhaltet die beiden Netzwerke für den Generator und den Diskriminator, die miteinander interagieren können. Bei der Implementierung des *Generative Adversarial Networks* habe ich mich an dem im TensorFlow Tutorial über *Deep Convolutional Generative Adversarial Networks* verwendeten Code orientiert.

5.1 Programmiersprache und Bibliotheken

Es gibt viele Programmiersprachen, mit denen man ein künstliches Neuronales Netzwerk implementieren und trainieren kann. Ich habe mich aus zwei Gründen für Python entschieden. Erstens besitze ich bereits genügend Programmierkenntnisse in dieser Sprache, um einen selbstlernenden Algorithmus zu schreiben. Und zweitens wird Python von Datenwissenschaftlern und Unternehmen am häufigsten verwendet, wodurch ich auf Programmierbibliotheken Zugriff habe, die mir mein Vorhaben erleichtern [16]. Eine Programmierbibliothek ist eine Sammlung von vorgefertigten Funktionen, die man benutzen kann, ohne sie vorher selbst programmiert zu haben. Ich werde in diesem Kapitel auf die einzelnen Bibliotheken und Hilfsprogramme kurz eingehen.

5.1.1 TensorFlow

TensorFlow ist ein von Google entwickeltes Programmiergerüst, das von Python-Programmen genutzt werden kann. Es wird zur datenstromorientierten Programmierung genutzt. In TensorFlow wird ein sogenannter Graph erstellt, durch den die Daten fortlaufend fliessen und durch mathematische Operationen verarbeitet werden. Künstliche Neuronale Netzwerke können durch Graphen dargestellt werden und ermöglichen effiziente Berechnungen, was schnellere Trainingszeiten zur Folge hat.

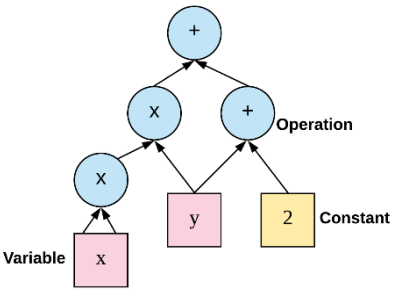

*Abb. 5.1: Beispiel eines Graphen für die Berechnung ((x * x) * y) + (y + 2).*

5.1.2 Keras

Keras ist eine benutzerfreundliche Programmierbibliothek, die das Erstellen von künstlichen Neuronalen Netzwerken erheblich erleichtert. Sie bietet Programmiermethoden, um Netzwerke zu definieren und diese zu trainieren. Der ganze Optimierungsprozess wird von Keras übernommen und kann durch das Aufrufen gewisser Methoden erreicht werden. Zuerst wird das Netzwerk durch ein *Sequential*-Objekt erstellt. Durch die *add*-Methode können dem Netzwerk einzelne Schichten hinzugefügt werden, um so die Struktur des Netzwerks zu definieren. Mit der *compile*-Methode wird das Optimierungsverfahren und die Fehlerfunktion ausgewählt und das Netzwerk erstellt. Als letztes wird das Netzwerk mit der *fit*-Methode trainiert.

```
1  # X- und Y-Werte der Funktion f(x)= 2x dienen als Trainingsdaten.
2  x = [0, 1, 2, 3, 4,  5,  6,  7,  8,  9]
3  y = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
4
5  # Netzwerk wird als "Sequential"-Objekt definiert.
6  netzwerk = keras.models.Sequential()
7  # Ein Dense Layer wird dem Netzwerk hinzugefügt. Diese
8  # Schicht hat ein Neuron (unit).
9  netzwerk.add(keras.layers.Dense(units=1, input_shape))
10
11 # Als Optimierungsverfahren wird eine modifizierte Version des
12 # Gradientenverfahrens (stochastic gradient descent) verwendet.
13 # Für die Fehlerfunktion dient die mittlere quadratische
14 # Abweichung (mean squared error).
15 netzwerk.compile(optimizer='sgd', loss='mse')
16
17 # Netzwerk wird 10000 mal optimiert.
18 netzwerk.fit(x, y, epochs=10000)
```

Abb. 5.2: Implementierung eines simplen künstlichen neuronalen Netzwerks mit Keras. Das Netzwerk lernt den linearen Zusammenhang zwischen den X- und Y-Werten der Trainingsdaten.

5.2 Datensatz

Damit ein künstliches neuronales Netzwerk lernen kann, werden natürlich auch Daten für das Training benötigt**.** In meiner Arbeit versuche ich, ein *Generative Adversarial Network* zu trainieren, das fotorealistische Bilder von menschlichen Gesichtern generieren kann. Darum sollten die Trainingsdaten aus Bildern von Gesichtern bestehen. Diese Bilder werden in einem Datensatz gespeichert.

Beim Erstellen des Datensatzes muss auf verschiedene Punkte geachtet werden:

1. Anzahl und Variation der Bilder: Möglichst viele verschiedene Personen, damit das Netzwerk ein verallgemeinertes Konzept eines Gesichtes lernen kann.
2. Dimension der Bilder: Die Höhe und Breite der Bilder müssen gleich sein.
3. Auflösung: Die Bilder sollten keine zu tiefe, aber auch keine zu hohe Auflösung besitzen. Ist die Auflösung zu tief, hat das Bild zu wenig Details und das Gesicht ist unerkennbar. Ist die Auflösung zu hoch, benötigt das Training sehr viel Rechenleistung und der Trainingsprozess dauert länger.

5.2.1 CelebFaces Attributes Datensatz

Nach einiger Recherche im Internet fand ich den vorgefertigten Datensatz *CelebFaces Attributes Dataset (CelebA)* von der Chinese University of Hong Kong. Dieser Datensatz besteht aus 200'000 Bildern mit Gesichtern von 10'000 verschiedenen Prominenten. Zusätzlich sind zu jedem Bild 40 Annotationen verfügbar, wie die Koordinaten-Position der Augen, die Gesichtsform oder ob die Person eine Brille trägt. Insgesamt ist dieser Datensatz 1.38 Gigabyte gross [17]. Für ein *Generative Adversarial Network* werden allerdings nur die Bilder ohne Annotationen benötigt.

Der Vorteil dieses Datensatzes ist die hohe Anzahl und Variation der Bilder. Jedes Bild wurde so zugeschnitten, dass es 178x218 Pixel misst. Ich habe die Grösse der Bilder für mein Netzwerk auf 64x64 Pixel verkleinert. Damit haben die Bilder eine Auflösung, bei der man das Gesicht noch erkennt und bei der mein Computer genug Rechenleistung besitzt, das Netzwerk zu trainieren.

Abb. 5.3: Bilder des CelebA Datensatzes (links) und die abgeänderten Bilder des Datensatzes für das Netzwerk.

5.2.2 Datensatz laden und vorverarbeiten

Der Datensatz muss zuerst geladen und vorverarbeitet werden, bevor er für das Training des *Generative Adversarial Networks* genutzt werden kann. TensorFlow besitzt einige Funktionen, mit denen die benötigten Bilder effizient geladen werden können, während das Netzwerks lernt. Als erstes wird der Datensatz durch die *load*-Methode geladen. Alle Bearbeitungen und Änderungen der Bilder werden in der *vorverarbeiten*-Funktion vorgenommen. Diese Funktion wird auf alle Bilder angewendet, die geladen werden. Da der Datensatz für jedes Bild über Annotationen verfügt, müssen als erstes die Bilder isoliert werden. Danach wird die Grösse der Bilder von 178x218 Pixel auf 64x64 Pixel geändert. Die Pixelwerte liegen zwischen von 0 und 256. Für ein besseres Training werden die Pixelwerte auf das Intervall von -1 bis 1 skaliert. Die Funktion wird durch die *map*-Methode des Datensatz-Objektes beim Laden auf die Bilder angewendet.

Als letztes wird die Batch-Grösse des Datensatzes festgelegt. Die Batch-Grösse gibt an, wie viele Bilder auf einmal geladen werden sollen.

```
# Datensatz "CelebA" wird geladen
datensatz = tfds.load(name='celeb_a', split='train', download=False)

def vorverarbeiten(example):
    bild = example['image'] # nur Bild (Image) wird verwendet.
    bild = tf.image.resize(bild, (64, 64)) # Grösse des Bildes wird auf 64x64 geändert
    bild = (bild / 127.5) - 1 # Daten werden von [0;256] auf [-1;1] skaliert.
    return bild

# Die vorverarbeiten Funktion wird durch die ".map"-Methode auf jedes Bild des Datensatzes angewendet,
# wenn es geladen wird.
datensatz = datensatz.map(vorverarbeiten, num_parallel_calls=tf.data.experimental.AUTOTUNE)
# Es werden immer 32 Bilder auf einmal geladen.
datensatz = datensatz.batch(batch_size=32)
```

Abb. 5.4: Der Datensatz wird geladen und eine vorverarbeiten-Funktion wird definiert und angewendet. Zuletzt wird die Batch-Grösse eingestellt.

5.3 Definition der Netzwerke

Die Netzwerke werden mit der Keras-Bibliothek einzeln definiert und später für das Training der Netzwerke zusammengesetzt. Als erstes wird der Generator erstellt. Die Ausgabe des Generator-Netzwerks soll ein Bild sein. Da ein RGB-Bild drei verschiedene Farbkanäle besitzt, soll das Netzwerk diese drei Farbkanäle ausgeben. Anstatt drei verschiedene Bilder zu generieren, wird ein dreidimensionales Bild erstellt. Länge und Breite entsprechen denen des Bildes, und die Tiefe kann als Anzahl an Kanälen verstanden werden. Die Ausgabe des Generators ist also ein dreidimensionales Objekt mit den Massen 64x64x3 Pixeln. Als Eingabe dient ein Vektor mit 100 zufälligen Werten, die unter der Normalverteilung generiert wurden.

Das Netzwerk besteht aus einer Eingabeschicht, gefolgt von drei Blöcken von Schichten. Ein Block besteht jeweils aus einem *Transposed Convolutional Layer*, einer *Batch-Normalisierung* und einer *ReLU-Aktivierung*. Ein *Transposed Convolutional Layer* ist ein umgekehrter *Convolutional Layer*. Die Funktionsweise ist dieselbe. Durch jeden Block wird die Länge und Breite des Bildes verdoppelt und die Tiefe halbiert.

```
# Eingabeschicht, Form: (100,) -> (4, 4, 256)
self.input_layer = Dense(units=4*4*256, input_shape=(eingabevektor_länge,))
self.reshape_1 = Reshape((4, 4, 256))

# Block 1, Form: (4, 4, 256) -> (8, 8, 128)
self.transposed_conv2d_1 = Conv2DTranspose(filters=128,kernel_size=5,
                                           strides=(2,2),padding='same')
self.batch_norm_1 = BatchNormalization()
self.relu_1 = ReLU()

# Block 2, Form: (8, 8, 128) -> (16, 16, 64)
self.transposed_conv2d_2 = Conv2DTranspose(filters=64, kernel_size=5,
                                           strides=(2,2), padding='same')
self.batch_norm_2 = BatchNormalization()
self.relu_2 = ReLU()

# Block 3, Form: (16, 16, 64) -> (32, 32, 32)
self.transposed_conv2d_3 = Conv2DTranspose(filters=32, kernel_size=5,
                                           strides=(2,2), padding='same')
self.batch_norm_3 = BatchNormalization()
self.relu_3 = ReLU()

# Ausgabeschicht, Form: (32, 32, 32) -> (64, 64, 3)
self.output_layer = Conv2DTranspose(filters=3, kernel_size=5, strides=(2,2),
                                    padding='same', activation='tanh')
```

Abb. 5.5: Struktur des Generatornetzwerks mit den einzelnen Schichten. Nach jedem Transposed Convolutional Layer wird die Bildlänge und- breite verdoppelt und die Tiefe halbiert.

Das Diskriminator-Netzwerk ist das Gegenstück des Generator-Netzwerks. Das Bild dient dieses Mal als Eingabeschicht, und die Ausgabeschicht gibt eine einzelne Zahl aus. Die Blöcke bestehen statt aus *Transposed Convolutional Layern* aus *Convolutional Layern* und aus ReLU-Aktivierungen, welche negative Werte nicht verkleinern, sondern ganz auf Null setzen.

```
# Eingabeschicht, Form: (64, 64, 3) -> (32, 32, 32)
self.input_layer = Conv2D(filters=32, kernel_size=5, strides=(2, 2),
                          padding='same', input_shape=(64, 64, 3))
self.leaky_0 = LeakyReLU(alpha=0.2)

# Block 1, Form: (32, 32, 32) -> (16, 16, 64)
self.conv2d_1 = Conv2D(filters=64, kernel_size=5, strides=(2, 2), padding='same')
self.batch_norm_1 = BatchNormalization()
self.leaky_1 = LeakyReLU(alpha=0.2)

# Block 2, Form: (16, 16, 64) -> (8, 8, 128)
self.conv2d_2 = Conv2D(filters=128, kernel_size=5, strides=(2, 2), padding='same')
self.batch_norm_2 = BatchNormalization()
self.leaky_2 = LeakyReLU(alpha=0.2)

# Block 3, Form: (8, 8, 128) - > (4, 4, 256)
self.conv2d_3 = Conv2D(filters=256, kernel_size=5, strides=(2, 2), padding='same')
self.batch_norm_3 = BatchNormalization()
self.leaky_3 = LeakyReLU(alpha=0.2)

# Ausgabeschicht, Form: (4, 4, 256) -> (1,)
self.flatten = Flatten()
self.output_layer = Dense(units=1, activation='sigmoid')
```

Abb. 5.6: Struktur des Diskriminator-Netzwerks mit den einzelnen Schichten.

5.4 Training des GAN

Nachdem das *Generative Adversarial Network* definiert wurde, kann es trainiert werden. Dabei wird der Generator und der Diskriminator abwechslungsweise trainiert. Wenn der Generator also optimiert wird, ändern sich die Parameter des Diskriminators nicht. Umgekehrt ändern sich die Parameter des Generators nicht, wenn der Diskriminator optimiert wird. Weiter wird für das Training eine Fehlerfunktion benötigt, die minimiert beziehungsweise maximiert werden soll.

5.4.1 Fehlerfunktion

Die Fehlerfunktion des Generators und Diskriminators bildet die Kreuzentropie. Wenn der Diskriminator die generierten Bilder fälschlicherweise als echt vorhersagt, ist die Fehlerfunktion des Generators tief. Denn das Ziel des Generators ist, den Diskriminator zu täuschen. Die Fehlerfunktion des Diskriminators besteht aus der Summe der Fehler beim Vorhersagen der echten und generierten Bilder. Kann der Diskriminator die Bilder richtig unterscheiden, ist die Fehlerfunktion tief.

```python
# Die Fehlerfunktionen Kreuzentropie
kreuz_entropie = tf.keras.losses.BinaryCrossentropy(from_logits=True)

def generator_fehler_funktion(vorhersage_fake_bilder):
    # Entropie zwischen der Vorhersage des Diskriminators bei generierten Bildern (fake) und dem Ziel-Wert Eins.
    return kreuz_entropie(tf.ones_like(vorhersage_fake_bilder), vorhersage_fake_bilder)


def diskriminator_fehler_funktion(vorhersage_echte_bilder, vorhersage_fake_bilder):

    # Entropie zwischen der Vorhersage des Diskriminators bei Bildern des Datensatzes (echt) und dem Ziel-Wert Eins.
    fehler_vorhersage_echte_bilder = kreuz_entropie(tf.ones_like(vorhersage_echte_bilder), vorhersage_echte_bilder)

    # Entropie zwischen der Vorhersage des Diskriminators bei generierten Bildern (fake) und dem Ziel-Wert Null.
    fehler_vorhersage_fake_bilder = kreuz_entropie(tf.zeros_like(vorhersage_fake_bilder), vorhersage_fake_bilder)

    # Summe bilden, um die gesamte Entropie zu berechnen
    gesamter_fehler = fehler_vorhersage_echte_bilder + fehler_vorhersage_fake_bilder

    return gesamter_fehler
```

Abb. 5.7: Fehlerfunktionen für Generator und Diskriminator.

5.4.2 Optimierungsalgorithmen des Generators und Diskriminators

Als Optimierungsverfahren für den Generator und den Diskriminator wird ein Algorithmus mit dem Namen *Adam Optimizer* verwendet. Dieser hat mehrere einstellbare Parameter, darunter die *Lernrate*. Mit der Lernrate wird die Stärke der Optimierung bestimmt. Ist die Lernrate hoch, werden die Gewichte der Netzwerke bei jedem Optimierungsschritt stärker geändert als bei einer tiefen Lernrate [18].

5.4.3 Trainingsschritt

Die Optimierung eines künstlichen neuronalen Netzwerks unter Anwendung des Gradientenverfahrens erfolgt iterativ. Es wird also die Optimierungsoperation immer wieder erneut angewandt. Diese Operation, auch *Trainingsschritt* genannt, berechnet die Fehlerfunktion für einen ausgewählten Teil des Datensatzes und verändert die Parameter der Netzwerke, sodass die Fehlerfunktion minimiert wird. Diesen Teil des Datensatz bezeichnet man als *Batch*. Ich habe für mein Netzwerk eine Batch-Grösse von 64 gewählt, also werden immer 64 Bilder, 32 vom Datensatz und 32 vom Generator, in einem Trainingsschritt verwendet, um die Netzwerke zu optimieren.

Bei einem *Generative Adversarial Network* werden während eines Trainingsschritts der Generator und der Diskriminator nacheinander einmal optimiert. Um den Fehler der beiden Netzwerke zu berechnen, müssen diese erst eine Vorhersage treffen. Also werden als erstes mithilfe des Generators 32 Bilder generiert. Danach werden diese 32 Bilder und weitere 32 Bilder des Datensatzes verwendet, damit der Diskriminator die Wahrscheinlichkeit für diese 64 Bilder vorhersagen kann, ob es sich um ein Bild des Datensatzes oder ein generiertes Bild handelt. Da nun sowohl die Vorhersage des Diskriminators als auch die wahren Labels bekannt sind, kann der Fehler durch die Fehlerfunktionen berechnet werden.

```
# Generator generiert 32 Bilder
eingabevektor = generator.generiere_eingabevektor(
                    batch_grösse=batch_grösse, # 32 von 64 Bilder werden von Generator generiert.
                    länge=100)

generierte_bilder = generator(eingabevektor)

# Diskriminator macht Vorhersage über die Wahrscheinlichkeit, dass die Bilder vom Datensatz stammen.
vorhersage_echte_bilder = diskriminator(datensatz_bilder) # 32 Bilder aus dem Datensatz
vorhersage_fake_bilder = diskriminator(generierte_bilder) # 32 Bilder von Generator
```

Abb. 5.8: Der Generator generiert Bilder und der Diskriminator trifft Vorhersagen über die Echtheit der Bilder.

Danach wird der Fehler des Diskriminators berechnet. Mit diesem Fehler kann der Gradient berechnet werden. Dieser Gradient gibt an, wie die Parameter des Diskriminators geändert werden, um dessen Fehlerfunktion zu minimieren. Dank TensorFlow kann dieser Gradient durch das Objekt *GradientTape()* automatisch berechnet werden. Der Funktion *GradientTape().gradient()* müssen die zu minimierende Variable, also der Fehler, und die Parameter des Diskriminators übergeben werden. Danach kann der Optimierer mit den Gradienten den Diskriminator optimieren, sodass der Fehler etwas geringer ist und die vorhergesagten Wahrscheinlichkeiten den wahren Labels mehr entsprechen.

```
# Diskriminator Optimierung
diskriminator_fehler = diskriminator_fehler_funktion(vorhersage_echte_bilder, vorhersage_fake_bilder)
diskriminator_gradienten = disk_tape.gradient(diskriminator_fehler, diskriminator.trainable_variables)
diskriminator_optimizer.apply_gradients(zip(diskriminator_gradienten, diskriminator.trainable_variables))

# Generator Optimierung
generator_fehler = generator_fehler_funktion(vorhersage_fake_bilder)
generator_gradienten = gen_tape.gradient(generator_fehler, generator.trainable_variables)
generator_optimizer.apply_gradients(zip(generator_gradienten, generator.trainable_variables))
```

Abb. 5.9: Das Gradientenverfahren wird angewendet, um den Diskriminator und den Generator zu optimieren.

5.4.4 Trainingsschleife

Da wir den Trainingsschritt des *Generative Adversarial Networks* nun definiert haben, können wir die gesamte Trainingsschleife programmieren. Das GAN wird nach einem Trainingsschritt nur sehr gering optimiert, also muss dieser oft wiederholt werden. Dies wird mit einer Schleife erreicht, in welcher der Trainingsschritt ausgeführt wird.

Das Training findet in sogenannten *Epochen* statt. Während einer Epoche wird der Trainingsschritt eine bestimmte Anzahl mal wiederholt. Die Anzahl Epochen und Trainingsschritte pro Epoche sowie andere Werte für die das Training werden *Hyperparameter* genannt. Die Hyperparameter haben einen Einfluss, wie schnell und gut das GAN trainiert wird. Diese werden ausserhalb der Trainingsschleife definiert, damit sie, wenn nötig, nur hier geändert werden müssen und überall im Code der geänderte Wert verwendet wird.

```
1  # HYPERPARAMETER
2  DATENSATZ_NAME = 'celeb_a'
3  BILD_GRÖSSE = (64, 64)
4  EPOCHEN = 50
5  TRAININGS_SCHRITTE_PRO_EPOCHE = 1024
6  BATCH_GRÖSSE = 64
7  GENERATOR_LERNRATE = 0.0005
8  DISKRIMINATOR_LERNRATE = 0.0004
```

Abb. 5.10: Hyperparameter des Generative Adversarial Networks.

Wie wir oben gesehen haben, werden Bilder des Datensatzes *CelebA* mit einer Grösse von 64x64 Pixel verwendet. Das Netzwerk wird 50 Epochen lang trainiert und während einer Epoche wird der Trainingsschritt 1024-mal wiederholt. Bei jedem Trainingsschritt werden 64 Bilder für einem Batch verwendet, um die Netzwerke zu optimieren. Die Lernrate des Generators beträgt 0.0005 und die des Diskriminators 0.0004. Insgesamt lernt das GAN von 50*1024*64 = 3'276'800 Bildern. Jedes Bild bekommt das GAN somit 8 Mal zu sehen.

Nach den Hyperparametern wird der Datensatz geladen:

```
# Datensatz laden
datensatz = datsensatz_laden_und_vorverarbeiten(
    name=DATENSATZ_NAME,
    batch_grösse=BATCH_GRÖSSE/2,
    bild_grösse=BILD_GRÖSSE)
```

Abb. 5.11: Funktion, die den Datensatz lädt und vorverarbeitet. Die Batch-Grösse des Datensatzes beträgt nur die Hälfte der gesamten Batch-Grösse, da beim Trainingsschritt nur 32 von 64 Bildern vom Datensatz verwendet werden. Die anderen 32 Bilder werden vom Generator erzeugt.

Darauf werden Netzwerke und die Optimierungsalgorithmen definiert:

```
# Netzwerke definieren
generator = Generator()
diskriminator = Diskriminator()

# Optimierungsverfahren definieren
generator_optimizer = tf.keras.optimizers.Adam(learning_rate=GENERATOR_LERNRATE)
diskriminator_optimizer = tf.keras.optimizers.Adam(learning_rate=DISKRIMINATOR_LERNRATE)
```

Abb. 5.12: Der Generator und der Diskriminator sowie die jeweiligen Optimierer werden definiert.

Dann erst erfolgt die Trainingsschleife, die den Trainingsschritt wiederholt:

```
def training(datensatz, epochen, batch_grösse, trainings_schritte_pro_epoche):

    for epoche in range(epochen):
        epoche += 1

        for batch in datensatz.take(trainings_schritte_pro_epoche):

            disk_fehler, gen_fehler = trainings_schritt(generator, diskriminator, batch,
                                                        batch_grösse=batch_grösse)

    return generator, diskriminator
```

Abb. 5.13: Trainings-Funktion, welche die Trainingsschleife beinhaltet. Der Trainingsschritt wird während einer Epoche eine bestimmte Anzahl mal wiederholt. Für den Trainingsschritt wird jeweils ein Batch aus dem Datensatz ausgewählt.

6 Ergebnisse und Verbesserungen

Der gesamte Code meines *Generative Adversarial Networks* sieht etwas komplexer aus, und ich habe zur besseren Übersicht einige Teile ausgelassen, die für das Verständnis des GAN nichts beitragen, sondern für das Programmieren benötigt werden.

Schon beim ersten Ausführen des Trainings entstand das erste Problem. Die Bilder blieben grau. Selbst nach längerem Training sah man keine Unterschiede.

6.1 Lineare statt sigmoidale Aktivierungsfunktion

Es stellte sich heraus, dass, wenn die Sigmoidfunktion für die Aktivierungsfunktion des Diskriminators verwendet wird, das Optimierungsverfahren nicht richtig funktionieren kann, da stark positive und negative Ausgabewerte immer auf Eins und minus Eins gesetzt werden. Die Gradienten können so schnell an Information verlieren und das Netzwerk wird nicht richtig optimiert. Nach Erfahrung kann das Problem behoben werden, wenn die Aktivierungsfunktion linear anstatt sigmoidal ist. Dadurch kann die Vorhersage des Diskriminators nun aber nicht mehr als Wahrscheinlichkeit angesehen werden.

Das Training dauerte für 50 Epochen ungefähr drei Stunden. Die generierten Bilder sahen nach dieser Trainingszeit schon einigermassen echt aus und die Gesichter waren klar zu erkennen.

Abb. 6.1: Generierte Bilder nach 25 Epochen (oben) und nach 50 Epochen (unten).

Doch bei längeren Trainingszeiten fing der Generator an, schlechter zu werden. Die Bilder wurden unerkennbar und zum Teil ganz schwarz. Dies lag daran, dass der Fehler des Diskriminators sehr gering im Vergleich zum Fehler des Generators war. Bei einem stabilen Training sollten die Fehler ungefähr gleich gross sein. Doch da der Diskriminator um einiges besser als der Generator wurde, konnte der Generator dem Diskriminator nicht mehr mithalten.

Abb. 6.2: Generierte Bilder nach 100 Epochen.

6.2 Trainingsbilder mit Rauschen

Das Problem des zu starken Diskriminators ist ein bekanntes Phänomen. Ich stiess in einem Forum auf den Vorschlag, Rauschen zu den Trainingsbildern hinzuzufügen. Dadurch wird es für den Diskriminator schwieriger, die echten von den generierten Bildern zu unterscheiden [19]. Diese Änderung hatte jedoch keinen grossen Einfluss auf das Training. Und die Bilder sahen nach 150 Epochen genauso undeutlich aus.

Abb. 6.3: Generierte Bilder nach 50 Epochen (oben) und nach 150 Epochen (unten). Den Trainingsbildern wurde Rauschen hinzugefügt.

7 Fazit

7.1 Selbstreflexion

Ich bin mit meiner Arbeit einigermassen zufrieden und ich denke, ich konnte die Grundidee eines *Generative Adversarial Networks* sowie generelle Einblicke in das maschinelle Lernen geben. Jedoch habe ich den Zeitaufwand unterschätzt. Das Training eines künstlichen neuronalen Netzwerks nimmt viel Zeit in Anspruch: man muss regelrecht experimentieren, um die passenden Hyperparameter zu finden.

Dazu kam die Schwierigkeit, dass das Gesamtthema der *Generative Adversarial Networks* noch sehr jung ist und damit wenig Literatur vorhanden ist. Es gibt aber eine grössere Community rund um das maschinelle Lernen, in der man sich gegenseitig mit Blogposts und Online-Tutorials hilft. Diese sind leider alle englischsprachig und es war schwierig, sich in das Thema einzulesen. Dadurch, dass ich mich aber schon länger mit dem maschinellem Lernen befasse, kannte ich mich schon etwas aus und wusste durchaus, wie ich an die notwendigen Informationen gelangen konnte. Am schwierigsten war das Verstehen der englischsprachigen Paper über *Generative Adversarial Networks*, da diese sehr kompliziert geschrieben sind und viele mathematische Schreibweisen verwenden, die mir nicht bekannt waren. Doch mit etwas Geduld und Durchhaltevermögen konnte ich die Paper genug verstehen, um sie als Quelle für meine Maturitätsarbeit zu benutzen.

7.2 Weiterführende Arbeit

Die Implementierung eines *Generative Adversarial Networks* ist mir zwar gelungen, aber ich möchte versuchen, durch weitere Techniken mein GAN zu verbessern, sodass Bilder mit einer höheren Auflösung sowie besserer Qualität der Gesichter generiert werden. Ausserdem möchte ich mein Netzwerk so modifizieren, dass ich bestimmen kann, ob die generierten Gesichter weiblich oder männlich sind oder ob die Person auf dem Bild eine Brille oder einen Hut trägt.

7.3 Danksagung

Ich möchte mich bei allen Personen bedanken, die mich bei meiner Maturitätsarbeit unterstützt haben. Besonders danke ich für die Betretung Herrn Michael Anderegg, der mir beim Zielesetzen und deren Einhaltung dieser geholfen hatte. Des Weiteren danke ich meinem Vater, der mir beim Lektorat der Arbeit geholfen hat.

Literaturverzeichnis

[1] Y. N. Harari, Homo Deus, München: C.H.Beck, 2017.

[2] „Wikipedia," [Online]. Available: https://de.wikipedia.org/wiki/Maschinelles_Lernen. [Zugriff am 27. Aug 2019].

[3] C. Duhigg, Die Macht der Gewohnheiten, München: Piper, 2012.

[4] D. Osinga, Deep Learning Kochbuch, Heidelberg: O'Reilly Verlag, 2019.

[5] „Wikipedia," [Online]. Available: https://de.wikipedia.org/wiki/K%C3%BCnstliches_Neuron#Bestandteile. [Zugriff am 30. Dez 2019].

[6] S. Doshi, „Medium.com," 13. Jan 2019. [Online]. Available: https://medium.com/@sdoshi579/optimizers-for-training-neural-network-59450d71caf6. [Zugriff am 31. Dez 2019].

[7] „Wikipedia," [Online]. Available: https://de.wikipedia.org/wiki/Gradientenverfahren. [Zugriff am 3. Sep 2019].

[8] „Wikipedia," [Online]. Available: https://de.wikipedia.org/wiki/Convolutional_Neural_Network. [Zugriff am 25. Nov 2019].

[9] P. Ramachandran, B. Zoph und Q. V. Le, „Searching for Activation Functions," https://arxiv.org/abs/1710.05941, 2017.

[10] A. Radford, L. Metz und S. Chintala, „Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks," https://arxiv.org/abs/1511.06434, 2016.

[11] B. Xu, N. Wang, T. Chen und M. Li, „Empirical Evaluation of Rectified Activations in Convolutional Network," https://arxiv.org/abs/1505.00853, 2015.

[12] S. Ioffe und C. Szegedy, „Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," https://arxiv.org/abs/1502.03167, 2015.

[13] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville und Y. Bengio, „Generative Adversarial Networks," https://arxiv.org/abs/1406.2661, 2014.

[14] Google Developers, „Generative Adversarial Networks," [Online]. Available: https://developers.google.com/machine-learning/gan/. [Zugriff am 12. Nov 2019].

[15] A. Géron, „A Short Introduction to Entropy, Cross-Entropy and KL-Divergence," 5. Feb 2018. [Online]. Available: https://www.youtube.com/watch?v=ErfnhcEV1O8.

[16] T. Elliott, „The GitHub Blog," GitHub, 24. Jan 2019. [Online]. Available: https://github.blog/2019-01-24-the-state-of-the-octoverse-machine-learning/. [Zugriff am 15. Dez 2019].

[17] „Multimedia Laboratory," The Chinese University of Hong Kong, Dezember 2015. [Online]. Available: http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html. [Zugriff am 25. Sep 2019].

[18] D. P. Kingma und J. Ba, „Adam: A Method for Stochastic Optimization," https://arxiv.org/abs/1412.6980, 2017.

[19] „G loss increase, what is this mean? ", GitHub. [Online]. Available: https://github.com/soumith/ganhacks/issues/14 [Zugriff am 18. Dez 2019].

Abbildungsverzeichnis

Bildnachweis

Abb. 3.1: https://blog.aimultiple.com/how-neural-networks-work/

Abb. 3.2: https://de.wikipedia.org/wiki/K%C3%BCnstliches_Neuron

Abb. 3.3: https://www.jeremyjordan.me/intro-to-neural-networks/

Abb. 3.4: https://www.freecodecamp.org/news/machine-learning-mean-squared-error-regression-line-c7dde9a26b93/

Abb. 3.5: https://www.freecodecamp.org/news/machine-learning-mean-squared-error-regression-line-c7dde9a26b93/

Abb. 3.6: https://www.javatpoint.com/convolutional-neural-network-in-tensorflow

Abb. 3.7: https://athemathmo.github.io/2016/04/29/convolutions-deep-learning.html

Abb. 3.8: https://de.wikipedia.org/wiki/Sigmoidfunktion

Abb. 3.9: S. loffe und C. Szegedy, „Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift“, https://arxiv.org/abs/1502.03167, 2015.

Abb. 4.1: https://developers.google.com/machine-learning/gan/gan_structure

Abb. 4.2: https://developers.google.com/machine-learning/gan/gan_structure

Abb. 4.3: https://stackabuse.com/introduction-to-gans-with-python-and-tensorflow/

Abb. 5.1: https://www.easy-tensorflow.com/tf-tutorials/basics/graph-and-session

Abb. 5.2 bis Abb. 5.13: Selbst erstellte Abbildungen

Abb. 6.1 bis Abb. 6.3: Selbst erstellte Abbildungen