# GAN - Generative Adversarial Networks

Eine weitere Architektur

Von

Luca Ritz

*BFH - Berner Fachhochschule*

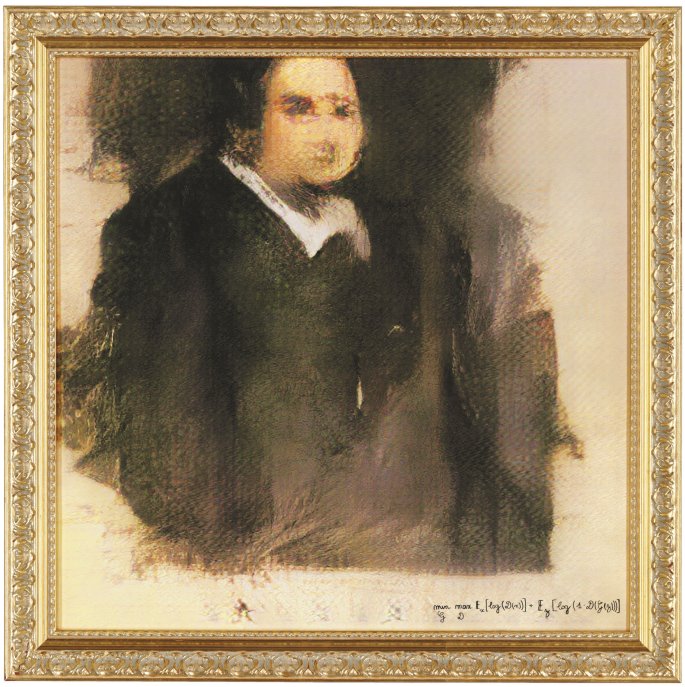

Abbildung 1: Portrait of Edmond Belamy [GAN18]

2021
PUBLISHER

# Inhaltsverzeichnis

# Kapitel 1

# Einführung

Am 25. Oktober 2018 wurde bei Christies das erste Gemälde verkauft, welches durch künstliche Intelligenz erschaffen wurde [GAN18]. Auf etwa 10'000 USD geschätzt, brachte es dem Anbieter doch über 432'500 USD ein. Es zeigt das Porträt einer fiktiven Person namens Edmond Belamy, trainiert mit über 15'000 Bildnissen verschiedener Epochen [Fel18]. Ob dieses „Gemälde" nun als kreatives Werk angesehen werden kann, ist sehr umstritten und wird auch nicht weiter behandelt. Es steht viel mehr der Künstler im Mittelpunkt. Der Künstler namens GAN.

Doch wieso ist so ein GAN überhaupt von Interesse? Leser mit Vorkentnissen im Bereich der künstlichen neuronalen Netzwerke haben wahrscheinlich schon gemerkt, dass es nicht trivial ist, neue Daten aufgrund von Bestehenden zu generieren. Für alle, welche nicht Wissen, was gemeint ist, sei gesagt, dass ein KNN Daten klassifizieren kann, aber nicht unbedingt neue Daten erzeugen. Das Generieren erfordert weitaus mehr. Es soll ein kleines Beispiel behandelt werden, um aufzuzeigen, was gemeint ist. Die Idee zu diesem Gedankenexperiment stammt von „Computerphile" [Com17]. Ein neuronales Netz lernt auf Basis von Input-Daten ein Modell. In diesem Fall gibt es deren drei Input-Daten, welche in der Abbildung 1.1 dargestellt werden. Wie erkannt werden kann, sind diese Datenpunkte sehr zufällig gewählt.

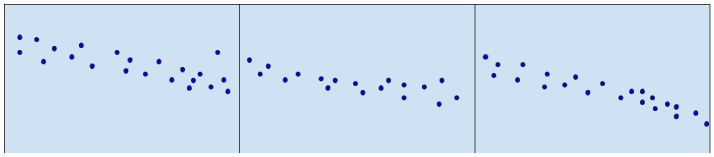

Abbildung 1.1: Input-Daten für KNN

Das Modell, welches das KNN lernt, kann wie in Abbildung 1.2 ersichtlich visualisiert werden. Es bildet eine möglichst gute Annäherung an alle Datenpunkte ab. Doch wie kann nun ein

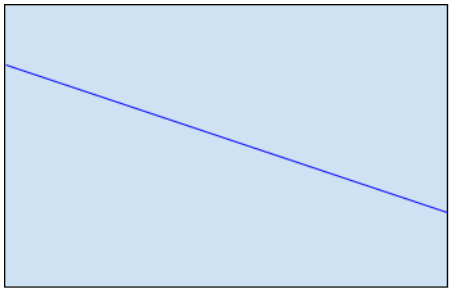

Abbildung 1.2: Gelerntes Modell

neues Sample, welches möglichst mit den Input-Daten korreliert, erzeugt werden? Das einzige, was das Netzwerk wahrscheinlich kann, sind zufällige Datenpunkte auf der Modellgeraden bestimmen. In der Abbildung 1.3 kann entnommen werden, dass dies nicht wirklich natürlich

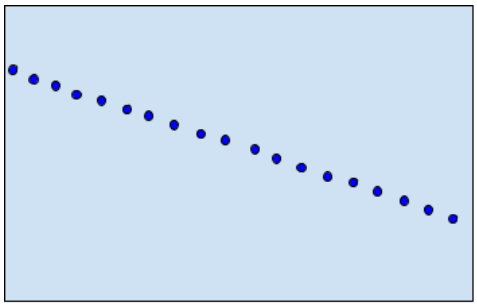

Abbildung 1.3: Generiertes Sample aufgrund von gelerntem Modell

wirkt und nicht mit den Input-Daten übereinstimmt. Die Frage stellt sich nun, wie also solche natürlich wirkenden Samples erzeugt werden können? Eine Antwort darauf liefert GAN. Die vorliegende Arbeit geht der Fragestellung nach, wie diese „Generative Adversarial Networks“ funktionieren.

# Kapitel 2

# GAN

Ein „Generative Adversarial Net“ besteht aus zwei Teilstücken. Der erste Teil wird „generative model $G$“ genannt und generiert auf Basis eines originalen Datensets neue Datensamples. Der zweite Teil nennt sich „discriminative model $D$“ und schätzt die Wahrscheinlichkeit, ob ein solches Sample, welches $G$ generiert, aus dem originalen Datenset stammt. $D$ selbst hat als Output also eine reele Zahl. [Cre+18] In den Worten der Autoren ausgedrückt ist ein GAN ein Spiel zwischen Geldfälschern und Polizisten: «*The generative model can be thought of as analogous to a team of counterfeiters, trying to produce fake currency and use it without detection, while the discriminative model is analogous to the police, trying to detect the counterfeit currency. Competition in this game drives both teams to improve their methods until the counterfeits are indistiguishable from the genuine articles*» [Cre+18]. Anders ausgedrückt versucht $G$ immer besser zu werden, um $D$ möglichst zu überlisten. $D$ hingegen sieht sich einem immer besser werdenden $G$ gegenüber gestellt und muss seine eigenen Schwachstellen ausbessern, um $G$ entgegenzuhalten. Eine Architektur, die so aufgebaut ist, dass sie sich gegenseitig immer weiter verbessert. Aus diesm Grund auch „Generative **Adversarial** Net“ genannt.

Im Endeffekt besteht das Ziel also darin, dass $D$ nicht mehr sagen kann, ob ein Sample von $G$ oder aus dem originalen Datenset stammt. Somit sollte $D$ für alle Samples immer $\frac{1}{2}$ ausgeben. Dies bedeutet, dass ein Sample dieselbe Wahrscheinlichkeit hat aus dem originalen Datenset oder aus dem generierten Set zu stammen und diese beiden Datensets daher nicht mehr unterscheidbar sind. Der Trainingsprozess für $G$ kann weiterhin als Maximierung der Fehlerwahrscheinlichkeit für $D$ angegeben werden. Dies entspricht also einem Zwei-Spieler Minimax-Algorithmus, auf welchen in dieser Arbeit ebenfalls noch eingegangen wird.[Cre+18]

Das generative Modell $G$ sowie das unterscheidende Modell $D$ bestehen jeweils aus zwei Multilayer-Perceptronen. Dies hat den Vorteil, dass die Modelle mittels Backpropagation trainiert werden können [Cre+18].

## 2.1 Neuronale Netze - Multilayer Perceptron

Da die Architektur des GAN auf zwei neuronalen Netzen beruht, wird in diesem Kapitel kurz darauf eingegangen. Dieser Einschub gibt einen groben Überblick, hat aber nicht die Absicht, ein vertieftes Wissen zu vermitteln.

Die neuronalen Netze sind, wie so manches, über die Zeit herangereift. Die Intention besteht nicht darin, das menschliche Gehirn nachzubauen, sondern eher eine Art Modell für die Informationsverarbeitung zu beschreiben [Unk20b]. Wie in der Biologie versucht man dies in diesen Netzen über Neuronen, wo auch der Name herrührt. Ein solches Neuron bekommt als Eingabe eine Summe von gewichteten Werten, welche es durch eine sogenannte „Aktivierungsfunktion“ in einen Ausgabewert umwandelt.

Am Anfang dieser Netzwerke stand das Perzeptron, welches nur ein einziges künstliches Neuron beinhaltet und von Frank Rosenblatt beschrieben wurde.[Unk20c] Danach hat man damit begonnen, diese Neuronen über Layer miteinander zu verbinden, wobei man den ersten Layer als Input-Layer, die Layer dazwischen als Hidden-Layer und den letzten Layer aus Output-Layer kennt. Der Output-Layer beinhaltet in einem einfachen KNN so viele Neuronen, wie es Klassen gibt. Jedes dieser Output-Neuronen gibt die Wahrscheinlichkeit an, ob eine Eingabe zu der jeweiligen Klasse gehört [Unk20b].

Ein solches neuronales Netz ist eigentlich eine Optimierungsaufgabe. In dem Fall geht es darum, am Ende eine Fehlerfunktion, welche den Abstand des Ist- und Sollzustands beschreibt, zu minimieren. Je geringer der Fehler, desto besser klassifiziert das Netz. Die Variablen, welche dabei angepasst werden können, sind die Gewichte, welche jeweils auf einer Verbindung zwischen zwei Neuronen liegen. Wie wahrscheinlich jeder aufmerksame Leser erkennt, werden dies sehr schnell sehr viele Variablen, weswegen normale Verfahren wie z.B. der Gradientenabstieg zwar eingesetzt werden können, aber sehr ineffizient sind. Der Durchbruch dieser Netzwerke wurde daher erst mit der Beschreibung des Backpropagation-Algorithmus erzielt, welcher ein Spezialfall des Gradientenabstiegverfahrens darstellt und die Gewichte aufgrund ihres Anteils am Fehler anpasst [Unk21d]. Heute gibt es sehr viele verschieden Arten und Architekturen von künstlichen neuronalen Netzen, auf welche nun nicht weiter eingegangen wird.

## 2.2 Funktionsweise

Formell ausgedrückt besteht ein GAN aus zwei ableitbaren Funktionen $G(z;\theta_g)$ und $D(x,\theta_d)$, welche durch zwei Multilayer-Perceptronen dargestellt werden. Dabei bilden jeweils $\theta_g$ sowie $\theta_d$ die Parameter der Netzwerke. Um die generierten Daten $p_g$ über ein Sample $x$ aus dem originalen Datenset $p_{data}$ zu trainieren, wird eine Input-Noise-Funktion $p_z(z)$ definiert. $G(z;\theta_g)$ bildet dabei das Mapping zwischen diesen Noise-Daten und dem originalen Datenset $p_{data}$. Aus dem Noise wird also ein Sample erzeugt, welches möglichst mit den Samples aus $p_{data}$ korreliert. $D(x)$ wiederum gibt einen einzelnen Skalar aus, der die Wahrscheinlichkeit beschreibt, ob ein Sample $x$ eher aus dem originalen Datenset $p_{data}$ als aus dem generierten Datenset $p_g$ (beinhaltet alle Samples von $G$) stammt. $D$ wird nun so trainiert, um die Wahrscheinlichkeit zu maximieren, das korrekte Label einem Input-Sample $x$ zu geben. Dabei gibt es zwei Labels: „Stammt aus original Datenset“ und „Stammt aus generiertem Datenset“. Simultan dazu wird $G$ trainiert, um die Funktion $\log(1 - D(G(z)))$ zu minimieren[Cre+18, p. 1]. In Worten: $G(z)$ generiert ein Sample aus den Noise-Daten. Davon wird die Wahrscheinlichkeit berechnet, ob dieses Sample aus dem originalen Datenset stammt. Wird dieser Wert gross, das heisst, $D$ hat das Gefühl, dass das generierte Sample aus dem originalen Datenset stammt, so geht der Wert innerhalb des Logarithmus gegen 0 und damit gegen $-\infty$. Dadurch wird also

über diese Minimierung erzielt, dass $G$ möglichst Daten generiert, die $D$ nicht dem generierten Datenset sondern dem originalen Datenset selbst zuweist. Die Autoren verweisen hier auf die Tatsache, dass die beiden neuronalen Netze $D$ und $G$ ein Zwei-Spieler minimax-Spiel mit der folgenden Value-Funktion $V(G,D)$ spielen [Cre+18, p. 2]:

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

Nachfolgend werden einige Gedanken des Autors dazu aufgeführt.
**Fall 1 - Discriminator macht alles richtig**:

$$D(x) = 1, D(G(z)) = 0 \Rightarrow \log(1) + \log(1-0) = \log(1) + \log(1) = 0 \tag{2.2}$$

**Fall 2 - Discriminator kann nicht mehr unterscheiden**:

$$D(x) = \frac{1}{2}, D(G(z)) = \frac{1}{2} \Rightarrow \log(\frac{1}{2}) + \log(1-\frac{1}{2}) = \log(\frac{1}{2}) + \log(\frac{1}{2}) = -\log(4) = -1.3862 \tag{2.3}$$

Nach den Autoren bildet der Fall 2 das globale Minimum, welches eben nur erzielt werden kann, wenn der Discriminator nicht mehr zwischen den beiden Datensets unterscheiden kann [Cre+18, p. 4-5]. Es gilt: $p_{data} = p_g$ Was genau diese sogenannte Value-Funktion $V(G,D)$ darstellt und wie sie hergeleitet wurde, wird im nächsten Abschnitt genauer erläutert.

Nun stellt sich also noch die Frage, wie denn genau $G$ lernt. Nach „Computerphile“ erhält $G$ Informationen von $D$. $G$ kennt also die Schwächen von $D$ und kann diese besser ausnützen [Com17]. Um dies besser zu verstehen, wurde ein Artikel hinzugezogen, welcher sich einem Beispiel in Tensorflow widmet [Unk21b]. Dort wird eine Loss-Funktion[1] sowohl für den Generator, wie auch den Discriminator angegeben. Die Loss-Funktionen werden im nachfolgenden einzeln besprochen. Zuerst wird sich dem Generator gewidmet.

```
def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)
```

Die Loss-Funktion nach [Unk21b] für den Generator besteht also im Wesentlichen aus einer Kreuzentropie[2] zwischen der Wahrscheinlichkeit des Diskriminators für die generierten Samples sowie der Wahrheit (alle Wahrscheinlichkeiten sind in dem Fall 1). Dies bedeutet, der Fehler für den Generator ist 0, wenn der Diskriminator alle generierten Samples als Sample aus dem originalen Datenset anerkennt.
Für den Diskriminator ist die Sache ein wenig komplizierter [Unk21b].

```
def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss
```

An dieser Stelle werden nun die einzelnen Zeilen der Gleichung 2.1 zugewiesen. Dazu muss bekannt sein, dass der Diskriminator ein binäres Problem beschreibt. Weiterhin scheint die

[1] Die Loss-Function bildet die Bewertung eines neuronalen Netzes. Sie gleicht eine Menge von Wahrscheinlichkeiten eines Ereignisses mit einer anderen ab und soll 0 ergeben, wenn die Mengen identisch sind.[Unk21c]

[2] Beschreibung in Anhang 4.1

Gleichung 2.1 der Kreuzentropie entnommen zu sein. Diese kann für einen binären Klassifizierer wie in Formel 4.14 geschrieben werden [Tir20] [3].

$$H(P,Q) = -\sum_{x \in X} (P(x) \cdot log(Q(x)) + (1 - P(x)) \cdot log(1 - Q(x))) \tag{2.4}$$

Die erste Zeile der „discriminator_loss“ Funktion berechnet die Kreuzentropie zwischen den Wahrscheinlichkeiten der originalen Samples sowie der erwarteten Wahrscheinlichkeiten von $D$ für die originalen Samples. Für den Diskriminator entspricht $P(x) = 1$ für alle Wahrscheinlichkeiten, wobei $Q(x) = D(x)$ gilt und das Sample $x$ aus $p_{data}$ stammt. Eingesetzt in Formel 4.14 ergibt dies 4.15.

$$H(P,D) = -\sum_{x \in p_{data}} (P(x) \cdot log(D(x)) + (1 - P(x)) \cdot log(1 - D(x))) \tag{2.5}$$

$$H(1,D) = -\sum_{x \in p_{data}} log(D(x)) \tag{2.6}$$

Die darauffolgende Zeile der Funktion „discriminator_loss“ wird dem zweiten Summand der Gleichung 2.1 zugewiesen. In diesem Fall ist $P(x) = 0$ für alle erwarteten Wahrscheinlichkeiten (das generierte Sample gehört nicht zum originalen Datenset), $Q(x) = D(x)$ und $x$ stammt aus $p_g$.

$$H(P,D) = -\sum_{x \in p_g} (P(x) \cdot log(D(x)) + (1 - P(x)) \cdot log(1 - D(x))) \tag{2.7}$$

$$H(0,D) = -\sum_{x \in p_g} log(1 - D(x)) \tag{2.8}$$

Wenn nun in Betracht gezogen wird, dass $x$ nicht aus $p_g$ stammt, sondern aus den Noise-Samples $p_z$ und $x = G(z)$ gilt, dann kann die Formel 4.16 so umgeschrieben werden, dass sie mit der Formel 2.1 korreliert.

$$H(0,D(G)) = -\sum_{z \in p_z} log(1 - D(G(z))) \tag{2.9}$$

Zusammenaddiert ergeben die Formeln 2.6 und 2.9 fast den Loss-Wert der Formel 2.1 und damit auch die Minimax-Value-Funktion, welche nach den Formeln 2.6 und 2.9 eigentlich von $D$ minimiert werden will.

$$\widetilde{V(D,G)} = -\sum_{x \in p_{data}} log(D(x)) - \sum_{z \in p_z} log(1 - D(G(z))) \tag{2.10}$$

$-1$ soll nun noch ausgeklammert werden, damit wird ersichtlich, dass es sich allem Anschein nach um die umgedrehte Loss-Function von $D$ handelt (mit kleinem Zusatz).

$$\widetilde{V(D,G)} = -(\frac{1}{|p_{data}|} \cdot \sum_{x \in p_{data}} log(D(x)) + \frac{1}{|p_z|} \cdot \sum_{z \in p_z} log(1 - D(G(z)))) \tag{2.11}$$

$$-\frac{1}{m} \widetilde{V(D,G)} = V(D,G) \tag{2.12}$$

Formel 2.12 erklärt nun den Zusammenhang zwischen der über die Kreuzentropie hergeleiteten Formel wie auch der Value-Function aus Formel 2.1. Für den Minimax-Algorithmus wird also die Richtung der Optimierung umgedreht, um nach $G$ zu minimieren. Zu beachten gilt auch, dass die Formel 2.1 nicht einfach die Summe bildet sondern das arithmetische Mittel, wobei $m$ die Anzahl der Samples ist. Dies gilt auch nur, wenn die beiden Datensets der Fomel 2.1 gleich gross sind und genau $m$ Samples beinhalten. Für eine genauere Erlätuerung kann der Algorithmus hinzugezogen werden.

[3] Für eine genauere Herleitung, siehe dazu den Anhang 4.1.1

### 2.2.1 Einschub - Minimax

Um zu verstehen, wieso es sich hierbei um einen solchen Minimax-Algorithmus handelt, wird in diesem Abschnitt ebendieser mit Verweis auf die Gleichung 2.1 eingeführt. Der Algorithmus geht auf den Versuch zurück, für ein Zwei-Spieler-Spiel eine schlaue KI zu erschaffen, die möglichst weit in die Zukunft sehen und abschätzen kann, was der Gegener tun wird. Dabei geht man davon aus, dass der Gegner immer so agiert, damit er seine Gewinnwahrscheinlichkeit maximiert und die der KI minimiert.[Unk19] Die Züge, welche gemacht werden können, werden über einen sogenannten Game-Tree dargestellt. Dieser ist in Abbildung 2.1 ersichtlich, es ist der Game-Tree für ein GAN gegeben. Jeder Layer stellt einen Zug in diesem Spiel dar. Innerhalb eines Zuges gibt es viele Möglichkeiten. Jede dieser Möglichkeiten wird über einen Node dargestellt. Für ein GAN sind dies alle möglichen Zustände der Variablen des neuronalen Netzwerks. Der Branching-Faktor ist also sehr gross. In der Gleichung 2.1 geht es um die Maximierung für $D$, weswegen die grünen Layer zu $D$ gehören. In diesen Zügen soll der Algorithmus die Value-Funktion $V(G,D)$ für einen Node aus seinen Nachfolgern maximal wählen. Hingegen soll $G$ minimiert werden, weswegen die roten Layer zu $G$ gehören. Innerhalb dieser Layer wird der minimale Wert der nachfolge States bevorzugt. Ist eine gewisse Tiefe erreicht worden, oder es gibt keine Nachfolge-States mehr, dann wird die Value-Funktion $V(G,D)$ für diese States aufgerufen und für den jeweiligen Status der Wert berechnet. Abbildung 2.1 zeigt einen möglichen ausgefüllten Game-Tree über drei Stufen hinweg. Der rote Pfad bildet nach

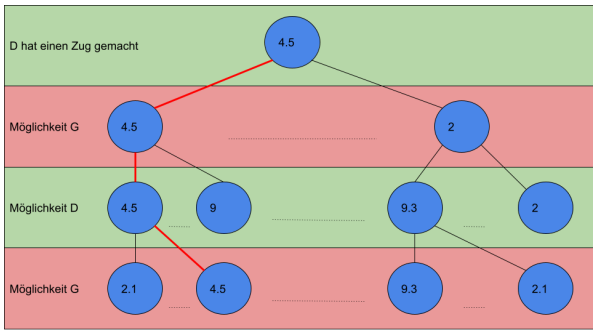


Abbildung 2.1: Berechneter GAN-Game-Tree

aktuellem Kentnisstand der beste Weg für $D$, und wird diesen wählen unter der Annahme, dass $G$ wiederum den Weg wählen wird, welcher für $G$ am besten ist.

## 2.3 Lernprozess

Der Algorithmus sieht vor, dass zuerst der Diskriminator $D$ trainiert wird. Dazu wird $k$-mal ein Minibatch aus den Noise-Samples wie auch aus den originalen Samples geladen. Der Diskriminator wird dann anhand der Formel 2.1 aktualisiert. Nach diesen $k$-Schritten wird wiederum ein Minibatch von Noise-Samples geladen und der Generator damit trainiert. Dieser Ablauf ist aus dem Algorithmus 1 ersichtlich, welcher von den Autoren des Papers beschrieben wird [Cre+18].

**for** *number of epochs* **do**
  **for** *k steps* **do**
    Sample minibatch of $m$ noise samples $\{z^{(1)}, \cdots, z^{(m)}\}$ from noise prior $p_g(z)$
    Sample minibatch of $m$ originals $x^{(1)}, \cdots, x^{(m)}$ from original data set $p_{data}(x)$
    Update the discriminator by ascending its stochastic gradient:
    $\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} [\log D(x^{(i)}) + \log(1 - D(G(z^{(i)})))]$
  **end**
  Sample minibatch of $m$ noise samples $\{z^{(1)}, \cdots, z^{(m)}\}$ from noise prior $p_g(z)$
  Update the generator by descending its stochastic gradient:
  $\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} [\log(1 - D(G(z^{(i)})))]$
**end**

**Algorithm 1:** Algorithmus

## 2.4 Noise zu Sample

Die Problematik der Bildgenerierung an sich wurde zu Beginn ein wenig angedeutet. Dies soll nun vertieft werden. Nach „Computerphile “ lernt ein Neuronales Netz die Klassifikation auf ein Label. Das heisst, es gibt jeweils genau eine richtige Lösung [Com17, t 4:00]. Die Problematik besteht nun darin, dass es in diesem Fall eine unendliche Menge an korrekten Lösungen gibt. Es kann also eine Lösung bei einem KNN angefragt werden, die erhaltene Lösung muss dann aber mit einer Art „Zufälligkeit“ verändert, respektive korrigiert werden.

Wie Eingangs erwähnt, bildet $G$ ein Mapper von Noise-Samples $z$ nach Samples in $x \in p_{data}$. Weiterhin ist nach dem Beispiel in Tensorflow das neuronale Netz von $G$ als Convolutional Neuronal Network definiert. Wahrscheinlich entspricht die Input-Dimension des Bilds der Output-Dimension [Unk21b]. Damit ist die Antwort auf die Frage bereits vornweg genommen. Die Frage selbst lautet, wie der Generator Bilder erzeugt. Um dies trotzdem ein wenig genauer zu erläutern wird zu Beginn ein neuronales Netz angenommen, das lediglich ein einzelnes Neuron in der Input-Schicht hat. Der Definitionsbereich dieses Neurons sei weiterhin auf 0 und 1 beschränkt, ist also binär. Dementsprechend kann der Generator logischerweise genau zwei Bilder generieren, da es sich um einen deterministischen Algorithmus handelt, welcher keinerlei Zufälligkeit zulässt.

Irgendwoher muss diese Zufälligkeit dem Generator zugeführt werden. Dies geschieht nun eben bei diesem Input-Layer. Bei einem klassischen KNN kann es also nicht einfach nur ein Neuron geben. Es gibt eine ganze Menge und der Wert der einzelnen Inputs kann zwischen bestimmten Intervallen liegen. Weiterhin werden alle Werte zufällig bestimmt, was einem Rauschen gleichkommt, wie in Abbildung 2.2 ersichtlich ist.

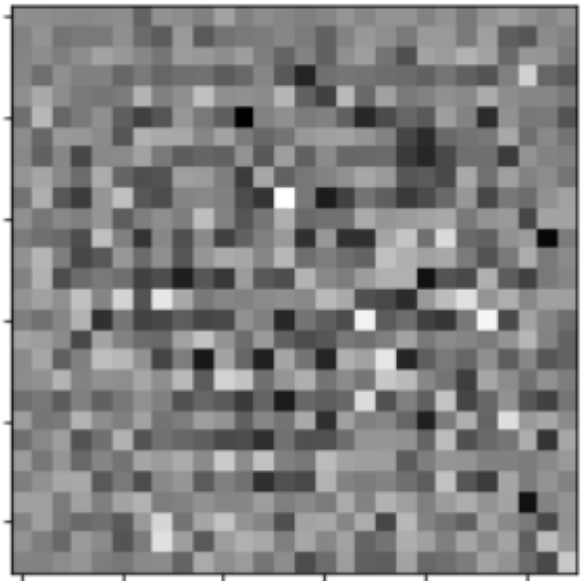

Abbildung 2.2: Input-Noise [Unk21b]

Was genau bedeutet nun dieses Rauschen? Nach „Computerphile“ kann dieses Rauschen einem Punkt in einem mehrdimensionalen Raum gleichgesetzt werden [Com17, t 16:45]. Wird dieser Punkt leicht angepasst, dann wird der Generator daraus ein leicht anderes Sample erzeugen, das einem Sample aus dem Datenset $p_{data}$ entspricht. Eine Richtung in diesem mehrdimensionalen Raum kann ebenfalls bestimmten Eigenschaften zugewiesen werden. Wird der Punkt z.B. in nur eine Richtung verschoben, so wird ein Objekt auf dem Output-Sample leicht grösser. Eine Richtung in diesem Raum würde also der Grösse entsprechen [Com17, t 17:35].

## 2.5 Vor- und Nachteile

Ein grosser Nachteil ist, dass $D$ und $G$ während dem Training synchronisiert werden müssen, $G$ kann also nicht häufig trainiert werden ohne auch $D$ zu aktualisieren. Ansonsten könnte das „Helvetica Szenario“ eintreten, wo $G$ sehr viele Samples aus dem Noise-Raum $p_z$ auf dasselbe Sample im originalen Raum $p_x$ abbildet [Cre+18].

Vorteile betreffen das Lernen an sich. So muss nur Backpropagation eingesetzt werden, um Gradienten zu finden. Es kann auch gut sein, dass statistische Vorteile erzielt werden durch die Tatsache, dass der Generator $G$ nicht direkt mit Samples aus den Testdaten sondern nur mit Informationen von $D$ aktualisiert wird. Die Autoren verweisen hier auf den Fakt, dass Komponenten der Testdaten nicht direkt die Parameter des Generators beeinflussen [Cre+18].

# Kapitel 3

# Fazit

GAN bietet einen sehr interessanten Ansatz, um neue Samples auf Basis von originalen Daten zu generieren. Die Idee, Input-Noise in den originalen Datenraum zu mappen und dies wiederum über zwei (fast) unabhängige neuronale Netze zu lösen, ist genial. Ein interessantes Anwendungsgebiet betrifft sicherlich das Machine-Learning selbst. Ein grosses Problem beim Training von solchen Netzen besteht bei den Daten, da meistens zu wenig vorhanden sind. Durch Augmentationsmethoden kann das Problem entschärft werden und nun ist in diesem Bereich wieder eine Möglichkeit dazu gekommen.

Auf jeden Fall ein interessanter Ansatz, den es sich lohnt weiterzuverfolgen.

# Kapitel 4

# Anhang

## 4.1 Loss-Function - Kreuzentropie

Diese Kreuzentropie lässt sich wie folgt beschreiben: «*Cross-entropy is a measure of the difference between two probability distributions for a given random variable or set of events*» [Jas19]. Es ist also ein Mass, um die Differenz zwischen zwei Wahrscheinlichkeitsverteilungen einer Zufallsvariable oder einer Menge von Ereignissen zu beschreiben. Um dies zu verstehen, wird zuerst die Entropie betrachtet. Diese beschreibt in der Informatik die Anzahl Bits, welche benötigt werden, um ein zufällig gewähltes Element aus einer Wahrscheinlichkeitsverteilung zu übertragen [Jas19]. Die Entropie kann weiterhin berechnet werden, wenn eine Menge an Ereignissen $x$ in $X$ sowie deren Wahrscheinlichkeit $P(x)$ bekannt sind. Dies ist aus der Gleichung 4.1 zu entnehmen.

$$H(P) = -\sum_{x \in X} P(x) \cdot log(P(x)) \tag{4.1}$$

Die Kreuzentropie baut nun auf dieser Definition auf. Sie beschreibt die Anzahl Bits, welche benötigt werden, um ein durchschnittliches Ereignis einer Wahrscheinlichkeitsverteilung im Vergleich zu einer anderen darzustellen [Jas19]. Nun kann ein Ereignis $x$ in zwei Wahrscheinlichkeitsräumen vorkommen. Einerseits in $P$, andererseits in $Q$. Weiterhin müssen die Wahrscheinlichkeiten der Ereignisse $P(x)$ und $Q(x)$ innerhalb dieser Verteilungen bekannt sein. Dadurch ergibt sich die Gleichung 4.2.

$$H(P,Q) = -\sum_{x \in X} P(x) \cdot log(Q(x)) \tag{4.2}$$

Wie genau ist dies nun im Kontext von Machine Learning zu verstehen? Grundsätzlich geht es darum eine Klassifikation vorzunehmen. Dabei kennt man für ein Sample die Wahrheit. Diese Wahrheit zeigt sich in der Form der Wahrscheinlichkeitsangabe für eine Klasse. Wenn es sich also z.B. um ein KNN handelt und dieses trainiert wird, um drei Klassen (Apfel, Birne, Zitrone) zu unterscheiden und ein Sample „Birne“ propagiert wird, dann ist die korrekte Wahrscheinlichkeitsverteilung $P$ (0, 1, 0), wohingegen das KNN für dieses Sample ein anderes Resultat liefert, nämlich $Q$ (0.2, 0.7, 0.1). Nun kann für das Ereignis „Birne“ die Kreuzentropie über beide Wahrscheinlichkeitsverteilungen berechnet werden, um den Unterschied derer zu

bestimmen. Konkret bedeutet das für das aktuelle Beispiel:

$$H(P,Q) = -(P(Apfel) \cdot log(Q(Apfel)) + P(Birne) \cdot log(Q(Birne)) + P(Zitrone) \cdot log(Q(Zitrone))) \tag{4.3}$$

$$H(P,Q) = -(0 \cdot log(0.2) + 1 \cdot log(0.7) + 0 \cdot log(0.1)) \tag{4.4}$$

$$H(P,Q) = -(1 \cdot log(0.7)) \tag{4.5}$$

$$H(P,Q) = -(-0.514573) \tag{4.6}$$

$$H(P,Q) = 0.514573 \tag{4.7}$$

Die Frage stellt sich nun, wieso diese Kreuzentropie sich so gut als Loss-Funktion eignet. Dazu muss bedacht werden, dass die Entropie für eine Wahrscheinlichkeitsverteilung wie im Falle von $P$ des vorherigen Beispiels immer 0 beträgt. Dies kann sehr einfach anhand der Entropie gezeigt werden. Man nehme für die Wahrscheinlichkeitsverteilung $P$ für die Ereignisse $x_1 \cdots x_n$ die folgenden Wahrscheinlichkeiten an: $(p_1, \cdots, p_k, \cdots, p_n)$ wobei $p_k = 1$ und $p_{i \neq k} = 0$ gilt. Dadurch ergibt sich für die Entropie folgenden Formel:

$$H(P) = -(\sum_{i=0}^{k-1}(P(x_i) \cdot log(P(x_i))) + P(x_k) \cdot log(P(x_k)) + \sum_{i=k+1}^{n}(P(x_i) \cdot log(P(x_i)))) \tag{4.8}$$

Aus der Formel 4.8 ist ersichtlich, dass die erste wie auch die letzte Summe auf 0 evaluieren, da die Wahrscheinlichkeit des Ereignisses $x_i$ jeweils 0 ist. Einzig das Ereignis $x_k$ ist 1, daher kann die Formel wie in 4.9 gekürzt werden.

$$H(P) = -(P(x_k) \cdot log(P(x_k))) \tag{4.9}$$

$$H(P) = -(1 \cdot log(1)) \tag{4.10}$$

$$H(P) = -(1 \cdot 0) \tag{4.11}$$

$$H(P) = 0 \tag{4.12}$$

Nun gilt es noch zu beachten, dass die Kreuzentropie der Entropie entspricht, wenn es sich um dieselbe Wahrscheinlichkeitsverteilung handelt. Dadurch wird klar, dass wenn ein Machine-Learning Algorithmus ein Resultat einer Klassifikation berechnet, welches exakt mit der Wahrheit übereinstimmt, dies 0 ergibt. Das Sample wurde korrekt klassifiziert.[Jas19].

### 4.1.1 Binäre Kreuzentropie

Besteht die Grundwahrscheinlichkeit $P(x)$ nur aus zwei möglichen Zuständen (0 oder 1), dann kann die Formel der Kreuzentropie vereinfacht werden. Zu Beginn sei angemerkt, dass zwei Wahrscheinlichkeiten angenommen werden können, welche zusammen 1 ergeben. In dem Fall $1 = P_1(x) + P_2(x)$. Da $P(x)$ binär ist, muss entweder $P_1(x)$ oder $P_2(x)$ 0 sein. Nun lässt sich $P_2(x)$ über die Gegenwahrscheinlichkeit von $P_1(x)$ ausdrücken. Siehe dazu die Formel 4.13

$$P_2(x) = 1 - P_1(x) \tag{4.13}$$

Über die binäre Eigenschaft von $P(x)$ lässt sich die Kreuzentropie in dem Fall wie in Formel 4.14 formulieren. Anzumerken sei hier, dass wenn $P_1(x) = 1$ gilt, nur der Logarithmus des Wahrscheinlichkeitswerts $Q_1(x)$ von Bedeutung ist. Wenn $P_2(x) = 1$ gilt, dann gilt automatisch $P_1(x) = 0$ und es ist nur noch der Logarithmus des Wahrscheinlichkeitswerts von $Q_2(x)$ relevant.

$$H(P,Q) = -\sum_{x \in X} (P_1(x) \cdot log(Q_1(x)) + P_2(x) \cdot log(Q_2(x))) \tag{4.14}$$

$P_1(x)$ sowie $Q_1(x)$ sind sehr natürlich, denn es ist klar, dass ein Sample $x$ eine Wahrscheinlichkeit in beiden Wahrscheinlichkeitsverteilungen hat. Es irritiert aber, dass dort anscheinend zwei Wahrscheinlichkeiten für ein und dasselbe Ereignis/Sample existieren sollen, also $P_2(x)$ und $Q_2(x)$. Diese Information wird nun tatsächlich nicht direkt in den Wahrscheinlichkeitsverteilungen $P(x)$ und $Q(x)$ abgelegt, durch die Binarität von $P(x)$ sind sie jedoch durch die Formel 4.13 indirekt über die Gegenwahrscheinlichkeit gegeben. Siehe dazu Formel 4.15.

$$H(P,Q) = -\sum_{x \in X} (P_1(x) \cdot log(Q_1(x)) + (1 - P_1(x)) \cdot log(1 - Q_1(x))) \tag{4.15}$$

Nun können in der Formel 4.15 noch die Indizes der Wahrscheinlichkeitsverteilungen weggelassen werden, womit die Endgültige Formel in 4.16 entsteht.

$$H(P,Q) = -\sum_{x \in X} (P(x) \cdot log(Q(x)) + (1 - P(x)) \cdot log(1 - Q(x))) \tag{4.16}$$

# Abbildungsverzeichnis

# Literatur


[Com17] Computerphile. *Generative Adversarial Networks (GANs) - Computerphile*. [Online; accessed 28.02.2021]. 2017. URL: https://www.youtube.com/watch?v=Sw9r8CL98N0.

[Cre+18] A. Creswell u. a. «Generative Adversarial Networks: An Overview». In: *IEEE Signal Processing Magazine* 35.1 (2018), S. 1–3. DOI: 10.1109/MSP.2017.2765202.

[Fel18] Felix Philipp Ingold. *Kunst oder bloss technischer Schmiereffekt? Das Porträt von Edmond Belamy – geschaffen von künstlicher Intelligenz*. [Online; accessed 26.02.2021]. 2018. URL: https://www.nzz.ch/feuilleton/kuenstliche-intelligenz-und-statt-autorschaft-ld.1435762.

[GAN18] GAN. *Portrait of Edmond Belamy*. [Online; Zugriff 24.02.2021]. 2018. URL: https://www.christies.com/lot/lot-edmond-de-belamy-from-la-famille-de-6166184/?from=salesummery&intObjectID=6166184&sid=18abf70b-239c-41f7-bf78-99c5a4370bc7.

[Jas19] Jason Brownlee. *A Gentle Introduction to Cross-Entropy for Machine Learning*. [Online; accessed 02.03.2021]. 2019. URL: https://machinelearningmastery.com/cross-entropy-for-machine-learning/.

[Tir20] Tirth Patel. *Can't understand the loss functions for the GAN model used in the tensorflow documentation*. [Online; accessed 02.03.2021]. 2020. URL: https://stackoverflow.com/questions/61488761/cant-understand-the-loss-functions-for-the-gan-model-used-in-the-tensorflow-doc.

[Unk19] Unknown. *Game Theory — The Minimax Algorithm Explained*. [Online; accessed 28.02.2021]. 2019. URL: https://towardsdatascience.com/how-a-chess-playing-computer-thinks-about-its-next-move-8f028bd0e7b1.

[Unk20a] Unknown. *Generative Adversarial Networks*. [Online; accessed 26.02.2021]. 2020. URL: https://de.wikipedia.org/wiki/Generative_Adversarial_Networks.

[Unk20b] Unknown. *Künstliches neuronales Netz*. [Online; accessed 28.02.2021]. 2020. URL: https://de.wikipedia.org/wiki/K%C3%BCnstliches_neuronales_Netz.

[Unk20c] Unknown. *Perzeptron*. [Online; accessed 28.02.2021]. 2020. URL: https://de.wikipedia.org/wiki/Perzeptron.

[Unk21a] Unknown. *Convolutional Neural Network*. [Online; accessed 05.03.2021]. 2021. URL: https://de.wikipedia.org/wiki/Convolutional_Neural_Network.

[Unk21b] Unknown. *Deep Convolutional Generative Adversarial Network*. [Online; accessed 02.03.2021]. 2021. URL: https://www.tensorflow.org/tutorials/generative/dcgan.

[Unk21c] Unknown. *Loss function*. [Online; accessed 05.03.2021]. 2021. URL: https://en.wikipedia.org/wiki/Loss_function.

[Unk21d] Unknown. *Multilayer perceptron*. [Online; accessed 28.02.2021]. 2021. URL: https://de.wikipedia.org/wiki/Backpropagation.

[Unk21e] Unknown. *Multilayer perceptron*. [Online; accessed 28.02.2021]. 2021. URL: https://en.wikipedia.org/wiki/Multilayer_perceptron.

# Glossar

**GAN** Generative Adversarial Networks - Zwei neuronale Netze, die ein Nullsummenspiel durchführen. Eines erstellt Kandidaten, das andere bewertet sie.[Unk20a].

**Multilayer-Perceptron** Ein Multilayer-Perceptron ist eine Klasse von künstlichen neuronalen Netzen. Es besteht aus mindestens drei Layern: dem Input-Layer, einem Hidden-Layer und einem Output-Layer. Vielfach wird es auch als „vanilla“ neuronales Netz bezeichnet.[Unk21e].

**KNN** Siehe Multilayer-Perceptron.

**Backpropagation** Die Backpropagation beschreibt ein Verfahren, um neuronale Netze zu trainieren. Es basiert auf dem mittleren quadratischen Fehler und gehört zu der Gruppe der überwachten Lernverfahren.[Unk21d].

**Convolutional Neuronal Network** Dabei handelt es sich um eine spezielle Architektur eines KNN. Es funktioniert nach dem Prinzip der Konvolution. Das heisst, die Gewichte auf den Kanten sind nicht flach, diese werden über Konvolutions-Filter definiert und viele Bildabschnitte teilen sich so die Gewichte.[Unk21a].