

Table of Contents

Table of Contents	1
3-D Datenrekonstruktion mit generative Adversarial Networks	3
1 <i>Andreas Wiegand</i>	
1 Einleitung	3
1.1 Objectives	4
1.2 Outline	5
2 Grundlagen und ähnliche Arbeiten	5
2.1 Künstliche Neuronale Netzwerke	6
2.1.1 Backpropagation Algorithmus	9
2.1.2 Momentum	10
2.1.3 Batch Normalisation	11
2.1.4 Ziel-Funktion	14
2.2 Datenformat	17
2.2.1 Woher stammen die Daten	17
2.2.2 Das Problem mit 3D-Data bei Machine Learning Ansätzen	17
2.3 Deep Learning und Informationstheorie	20
2.4 Convolution Neural Networks	20
2.5 Autoencoder	24
2.5.1 Zielfunktion 3D-Punktwolken Autoencoder	25
2.6 Generative Adversarial Network	28
2.6.1 Probleme mit Generative Adversarial Networks	31
2.6.2 Lösungsansätze für Generative Adversarial Networks	
Probleme	32
2.6.3 Conditional Adversarial Networks	33
2.7 Conditional-GAN	34
2.8 3D-GAN	34
3 Methoden	36
3.1 Aufbau	36
3.1.1 Datensatz 1. Versuchsaufbau	36
3.1.2 Datensatz 2. Versuchsaufbau	36
3.1.3 Trainingsaufbau 1 - GAN	39
3.1.4 Trainingsaufbau 2 - CGAN für	
Punktwolkenrekonstruktion	39
3.2 Ergebnis	43
3.2.1 Ergebnisse - Versuchsaufbau 1	43
3.2.2 Ergebnisse - Versuchsaufbau 2	43
4 Evaluation und Ergebnisse	43
5 Zusammenfassung und Diskussion	43
Abbildungsverzeichnis	44

Tabellenverzeichnis	45
----------------------------------	----

Literaturverzeichnis	46
-----------------------------------	----

KDE K Desktop Environment

SQL Structured Query Language

Bash Bourne-again shell

JDK Java Development Kit

VM Virtuelle Maschine

CGAN Conditional Adversarial Networks

PIX2PIX Image-to-Image GAN

3-D Datenrekonstruktion mit generative Adversarial Networks

Andreas Wiegand

Master These in künstlicher Intelligenz

Zusammenfassung. Generative Adversarial Network(GAN) ist ein künstliches neuronales Netzwerk(ANN) aus dem Bereich der generativen Modelle. Die Aufgabe des GANs ist es, die Wahrscheinlichkeitsverteilung von Trainingsdaten zu erlernen und dadurch anschließend neue Samples aus dieser Wahrscheinlichkeitsverteilung zu generieren. Man erhofft sich, den hohen Datenaufwand beim trainieren von ANN zu umgehen und durch GANs neue Trainingsdaten zu generieren. Ziel dieser Arbeit ist es das Konzept von GANs auf 3D-Scans von Tabakblättern anzuwenden um die Wahrscheinlichkeitsverteilung von 3D-Daten zu erlernen. Ein weiteres Ziel ist es heraus zu finden ob mit Hilfe von GANs es möglich ist 3D-Daten welche beispielsweise beim Scannen unvollständig sind ergänzen zu können. Im Folgenden wird auf die Theorie von GANs eingegangen, wie deren Aufbau und deren zugrunde liegendes mathematische Modell. Anschließend werden die Methoden des Experimentes präsentiert sowie die Ergebnisse diskutiert.

1 Einleitung

Um genauere Prognosen über Erntemenge und frühzeitiger Erkennung von Krankheiten zu treffen werden 3D-Scan verfahren eingesetzt welche Pflanzen scannen und damit 3D-Daten liefern welche zur weiterverarbeitung von Machine-Learning Ansätzen benötigt werden. [EVLUT BILD EINBAUEN UND GENAUER AUF SCANVERFAHRENE INGEHEN]. Jedoch sind diese Scanverfahren wie jede Informationsübertragung mit sogenannten rauschen verbunden das heißt die Information vom Empfänger zum Sender wird verändert und behält nicht die ursprünglichen Inhalt. Beispiele dafür waren ein Blatt verdeckt ein anderes Blatt und lässt die Sensoren des Scanner nicht zu das unter trunder liegende Blatt zu erfassen. Um nun diesen Verlust von Daten zu verhindert muss geprüft werden ob die Möglichkeit besteht diese Daten zu reparieren in dem sie ergänzt werden. In dieser Arbeit wird ein Ansatz überprüft welcher es Möglich machen kann dieses Verhalten zu erhalten. Ein Ansatz der dafür Verwendet werden kann ist Deep Learning bei dem man das Deep Learning Modell den unterschied zwischen Reparierten und nicht reparierten Daten erlernt kann es rückschlüsse zeihen und selber Daten reparieren.

Deep learning, however, gained much popularity across many academic disciplines in recent years and has been used in computer vision successfully to

produce state-of-the-art results for various tasks. It is described as the application of multiple processing layers to produce multiple levels of representation. It is therefore capable of learning higher level representations of raw data, that can be used for the intended task, e.g. classification of an image. The most common realization are artificial neural networks (ANN) and especially convolutional neural networks (CNN) for processing data with a grid-like topology, e.g. images. Several publications have shown the effectiveness of CNNs for instance segmentation of plant leaves on images (e.g. Ren and Zemel, 2016). But there is a lack of research in applying deep neural networks to 3D representations of plants.

In den letzten Jahren haben sich im Deep Learning Bereich besonders die discriminativen Modelle hervorgehoben, welche Input Daten wie Bilder, Audio oder Text Daten zu bestimmte Klassen zuordnen. Der Grund für das steigende Interesse liegt in der niedrigen Fehlerrate bei discriminativen Aufgaben, im Vergleich zu anderen Maschine Learning Ansätzen, wie Decision Trees oder Markov Chains(?, ?). Die Modelle lernen eine Funktion welche Input Daten X auf ein Klassen Label Y abbildet. Die Modelle werden dabei von ANN repräsentiert. Man kann auch sagen das Model lernt die bedingte Wahrscheinlichkeitsverteilung $P(Y|X)$ (?, ?). Generative Modelle haben die Aufgabe die Wahrscheinlichkeitsverteilung von Trainingsdatendaten zu erlernen. Es lernt eine multivariate Verteilung $P(X, Y)$, was dank der Bayes Regel auch zu $P(Y|X)$ umgeformt werden kann und somit kann das Modell auch für discriminativen Aufgaben herangezogen werden. Gleichzeitig können aber neue (x,y) Paare erzeugt werden, was zu dem Ergebnis von neuen Datensätzen führt welche nicht Teil der Trainings-sample sind (?, ?). In diesem Paper wird speziell auf GAN, aus der Vielzahl von generativen Modellen eingegangen. Diese wurden von Goodfellow(?, ?) entwickelt und ebneten den Weg für Variationen, welche auf der Grundidee von GANs aufbauen. 2017 wurden alleine 227 neue Paper zu diesem Thema vorgestellt. Ein Grund weshalb GAN an Popularität gewinnt ist der, dass neuronale Netzwerke mit Zunahme der Trainingsdatenanzahl eine Erhöhung der Performance für die sie Trainiert werden zeigen. Was bedeutet,dass sich mit Zunahme der Datenanzahl die Chance auf eine bessere Performance der Neuronalen Netzwerke ergibt (?, ?). An diesem Punkt erhofft man sich durch GANS mehr qualitative Daten zu erzeugen und somit das Trainingsergebnis zu discriminativen Modelle zu verbessern.

1.1 Objectives

In den letzten Jahren haben sich im Deep Learning Bereich besonders die discriminativen Modelle hervorgehoben, welche Input Daten wie Bilder, Audio oder Text Daten zu bestimmte Klassen zuordnen. Der Grund für das steigende Interesse liegt in der niedrigen Fehlerrate bei discriminativen Aufgaben, im Vergleich zu anderen Maschine Learning Ansätzen, wie Decision Trees oder Markov Chains(?, ?). Die Modelle lernen eine Funktion welche Input Daten X auf ein Klassen Label Y abbildet. Die Modelle werden dabei von ANN repräsentiert.

Man kann auch sagen das Model lernt die bedingte Wahrscheinlichkeitsverteilung $P(Y|X)$ (? , ?). Generative Modelle haben die Aufgabe die Wahrscheinlichkeitsverteilung von Trainingsdatendaten zu erlernen. Es lernt eine multivariate Verteilung $P(X, Y)$, was dank der Bayes Regel auch zu $P(Y|X)$ umgeformt werden kann und somit kann das Modell auch für discriminativen Aufgaben herangezogen werden. Gleichzeitig können aber neue (x,y) Paare erzeugt werden, was zu dem Ergebnis von neuen Datensätzen führt welche nicht Teil der Trainings-sample sind (? , ?). In diesem Paper wird speziel auf GAN, aus der Vielzahl von generativen Modellen eingegangen. Diese wurden von Goodfellow(? , ?) entwickelt und ebneten den Weg für Variationen, welche auf der Grundidee von GANs aufbauen. 2017 wurden alleine 227 neue Paper zu diesem Thema vorgestellt. Ein Grund weshalb GAN an Popularität gewinnt ist der, dass neuronale Netzwerke mit Zunahme der Trainingsdatenanzahl eine Erhöhung der Performance für die sie Trainiert werden zeigen. Was bedeutet,dass sich mit Zunahme der Datenanzahl die Chance auf eine bessere Performance der Neuronalen Netzwerke ergibt (? , ?). An diesem Punkt erhofft man sich durch GANS mehr qualitative Daten zu erzeugen und somit das Trainingsergebnis zu discriminativen Modelle zu verbessern.

1.2 Outline

Diese Arbeit ist folgender Maßen strukturiert. In Kapitel 2 "Grundlagen und ähnliche Arbeiten" werden theoretische Grundlagen welche für diese Arbeit benötigt wird genauer beleuchtet. Außerdem sollen vorrangingen Arbeiten welche einfluß auf diese Ausüben vorgestellt werden um zu veranschaulichen aus welchen Gründen diese Funktionieren kann.

Kapitel 3 - stellt die Arbeit an sich vor. Welche Methoden gewählt wurden ob diese zu Entwickeln und Auszuführen. Die Trainingsdaten werden vorgestellt und Modelle aufgezeigt.

Kapitel 4 - Gibts ausführliche Angaben über die Ergebnisse von den Experimenten und evaluiert sie für ihre Aussagekraft.

Kapitel 5 - Gibt eine Zusammenfassung über die Arbeit und wird sie kritisch Diskutieren. Desweiteren wird ein Ausblick erstellt inwiefern das Ergebnis für zukünftige Arbeiten von relevanz ist.

2 Grundlagen und ähnliche Arbeiten

Im folgenden Kapitel wird auf theoretische Grundlagen eingegangen, welche zum Verständnis für Arbeit benötigt werden. Zunächst werden generative Modelle im Allgemeinen vorgestellt, welche den Grundgedanken der Datengeneration für GANs aufzeigen. Darauf folgend werden Convolutional Neuronale Netzwerke vorgestellt, aus welchen GANs aufgebaut werden können, um mit Bild Daten zu arbeiten.

2.1 Künstliche Neuronale Netzwerke

Künstliche Neuronale Netzwerke(KNN) ist eine Datenstruktur welche von biologischen neuronalen Netzen wie sie bei Lebewesen vorkommen inspiriert sind. KNN haben das Ziel ein Funktion f^* zu approximieren. Dabei werden Parameter Θ eines Models so angepasst, um die Abbildung von $y = f(x;\Theta)$ zu approximieren. Die Modelle werden auch als feedforward Neuronale Netzwerke betitelt weil der Informationsfluss des Models von Input zu Output fließt und keine Rekursion von Output zu Input statt findet. KNN können aus mehreren Schichten sogenannten Hidden Layer n besteht welche als $f^{(n)}$ dargestellt werden und n $i=1$ sein muss. Ein 2-Layer KNN ist dann definiert durch $f(x)=f^{(2)}(f^{(1)}(x))$. Es gibt den Input Layer welche den Input in das Netzwerk aufnimmt. Hat man ein ANN mit mehreren Layern spricht man auch von Deep Learning. Im vorherigen Beispiel ist dies $f^{(1)}$ und der letzte Layer des Netzwerkes wird Output Layer genannt. Im vorherigen Beispiel wäre das $f^{(2)}(?, ?)$. In 2.2 sind ein ANN exemplarisch dargestellt. Es soll die Konnektivität der einzelnen Layer veranschaulichen und den Informationsfluss von Input zu Output.

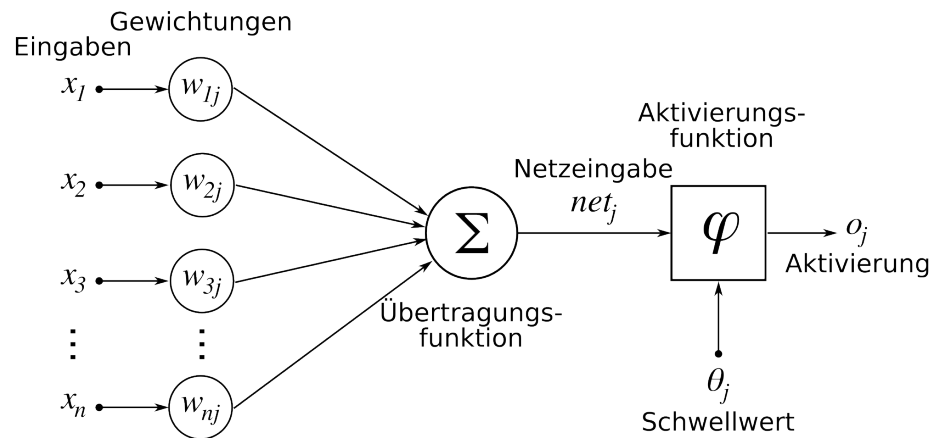


Abb. 1. künstliches Neuron

Jeder Layer besteht aus künstliche Neuronen diese haben ihren Namensgebung von aus der Natur stammende Neuron in Gehirnen von Lebewesen. Neuronen sind die Bausteine aus denen die Gehirne von Lebewesen, wie Fische, Vögel, Säugetiere zusammen gesetzt sind. Neuronen oder auch Nervenzellen bestehen in eine Zellkern der Zentrum der Zelle um sie herum sind Dendriten. Neuronen sind untereinander mit den Axon verbunden, diese haben an den enden Synapsen welche die grenze von Axon zur Nervenzelle einen Spalt bilden. Dieser Spalt kann überwunden werden indem von der Synapse Botenstoffe abgesendet werden welche sich dann an Rezeptor der gegenüberliegenden Synapse anhaften. Diese

Übertragung findet statt wenn an der Synapse ein bestimmter Schwellenwert überschritten hat wurde von elektrischen reizen welche die Zelle abfeuert lässt. Künstliche Neuronen haben diese Schwellenwert durch sogenannte Gewichte w_{ij} diese sind auf den Verbindungen zwischen den Neuronen in den unterschiedlichen Layern im Netzwerk.

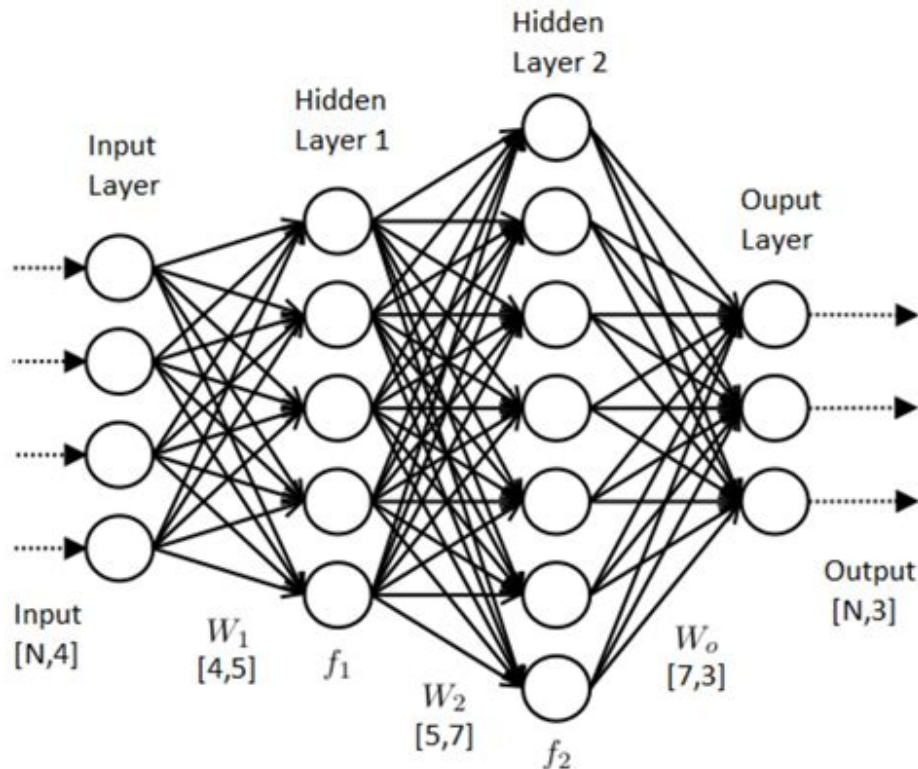


Abb. 2. künstliches neuronales Netzwerk

Jeder Layer eines KNN besteht aus mehreren Neuronen. Ein Neuron θ kann mehrere Input X_n erhalten und produziert einen Output ω . Die Berechnungen welche von den Neuron durchgeführt wird ist zunächst jeden Input X_n mit einem Gewicht w_{ij} zu multiplizieren wobei. Anschließend wird die Summe von $x \cdot w$ gebildet. Das Ergebnis wird dann in eine Aktivierungsfunktion gegeben. Die Aufgabe dieser ist es eine nicht lineare Transformation des Inputs zu erzeugen. Damit kann das KNN nicht lineare Funktionen zu lernen und somit komplexe Aufgaben zu lösen. Es gibt unterschiedliche Aktivierungsfunktionen Beispiele sind Sigmoid oder Softmax welche aus Abb. 17 und 17 entnommen werden können als Aktivierungsfunktion geeignet.

$$y_k = \text{Aktivierungsfunktion}(\sum_{j=0}^m (w_{kj} + x_j) + b_k)$$

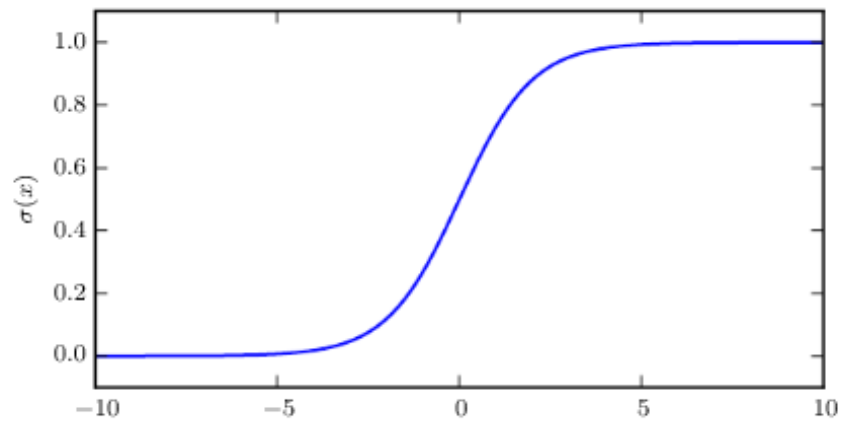


Abb. 3. Sigmoid Funktion

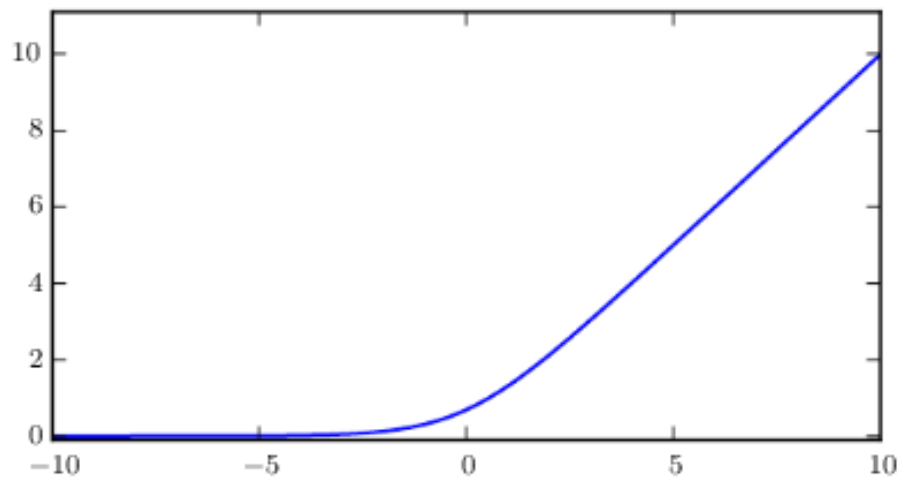


Abb. 4. Softmax

2.1.1 Backpropagation Algorithmus Um nun ANNs zu trainieren und den gewünschten Output y generieren wird der Backpropagation Algorithmus hergenommen. Dieser wurde von Geoffrey Hinton im Jahre 1989 vorgestellt(?, ?). Er konnte zeigen das sein Algorithmus schneller und effizienter auf Neuronale Networks arbeitet als andere vor ihm. Das mathematische Zugrundeliegende Konzept ist die der Partitiellen Ableitung von $\frac{\partial Z}{\partial w}$, wobei Z die Zielfunktion und w die Gewichte im zu Optimierenden Neuronale Netzwerk sind. Für eine Funktion $f(x) = y$ ist die Ableitung Definiert als $f'(x)$ oder $\frac{dy}{dx}$ und gibt die Steigung der Funktion an Punkt x an. Dieses Verfahren kann dabei helfen Funktionen zu optimieren. Wenn man weiß wie sehr die Steigung and Punkt x ist kann man x mit einer Änderung von der Ableitung x dahin gehen optimieren (?, ?). Wenn nun $f'(x) = 0$ haben wir keinerlei Information über die Steigung erreicht. Dies ist aber kein garant dafür das f ein Optimum erreicht hat. Es könnten wie in Abbildung unten dargestellt. Ein locales Minimum sein. Dass heißt das wir nur an diesen Punkt ein Minimum erreich thaben aber im Funktionsverlauf ein noch niedrigeres Minimum vorhanden ist. Oder einen Sattelpunkt welche ein Übergang zu einen anstieg der Funktion bildet. Ist nun die die Funktion f definiert als $f: \mathbb{R}^n \rightarrow \mathbb{R}$ und hat als Input mehrere Variablen. Die partielle Ableitung $\frac{\partial f(x)}{\partial x_i}$ zeigt an wie sehr sich $f(x)$ ändert wenn wir x_i ändern. Der Gradient $\nabla_x f(x)$ ist ein Vektor welche alle partiellen Ableitungen von f enthält. Nun kann man f optimierne in dem man in die Richtung des Gradienten geht welche negativ ist dieses Verfahren wird gradient descent genannt (?, ?).

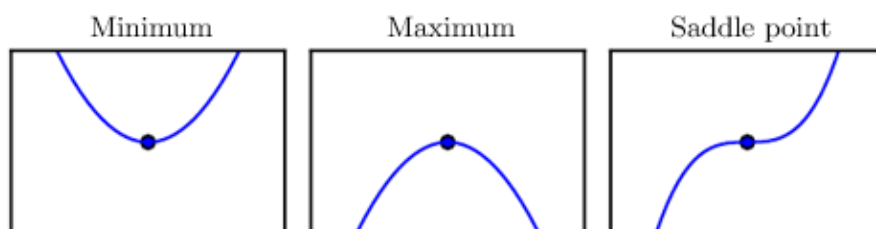


Abb. 5. Minimum Maximum Saddle Point

Algorithmen die für Deep Learning eingesetzt werden, beinhalten eine Art von Optimierung, ohne diese ist eine Umsetzung des Lernprozesses nahezu unmöglich. Diese Minimierungs-funktionen oder auch cost function (welche in vielen Publikationen unterschiedlich bezeichnet wird „loss“- oder „error function“) wollen immer dasselbe Ziel: eine gewisse Target Funktion ermitteln, welche für einen Input den gewünschten Output ausgibt. Das Ziel ist in anderen Worten eine Menge an Gewichten und Biases zu finden, für welche, die quadratischen Kosten ($C(w,b)$) so gering wie möglich sind. Um dies zu erreichen, wird der Gra-

der Decient Algorithmus eingesetzt (Nielsen, 2017). Der Grund des Einsatzes dieses Algorithmus ist derer, dass zwar versucht werden könnte die Anzahl der richtig klassifizierten Bilder direkt zu erhöhen. Aber das Problem ist, den Performancegewinn bei Veränderungen der Gewichte festzustellen. Da meisten kleine Veränderungen an den Gewichten und Baises führen zu keinerlei Veränderung bei der Klassifizierung. Somit liegt eine gewisse Schwierigkeit, bei der Ermittlung der richtigen Anpassung dieser Werte. Zuerst muss die quadratische cost function ($C(w,b)$) minimiert werden, bevor sich die Maximierung der Bestimmungsgenauigkeit des Netzwerkes als Ziel gesetzt kann (Nielsen, 2017).

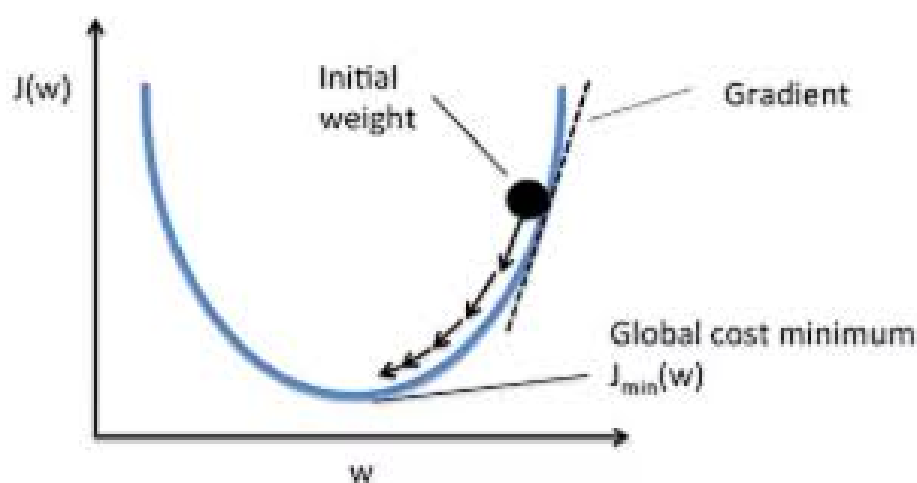


Abb. 6. LossFunktion

Backpropagation Algorithmus wird das Verfahren genannt mit den ANN trainiert werden. Dabei wird der Gradient der Ziel Funktion bestimmt.

2.1.2 Momentum Momentum hat das Ziel das Gradientenverfahren zu beschleunigen um eine effizienter Konfegung der Zielfunktion herbei zu führen und ein schnelleres Lernen erzielt.

$$v_{t+1} = \mu v_t - \epsilon \nabla f(\theta_t)$$

$$\theta_{t+1} = \theta_t + v_{t+1}$$

wobei $\epsilon > 0$ die Lernrate ist und $\mu \in [0,1]$ das Momentum ist und $\nabla f(\theta_t)$ der Gradient von θ_t ist. Je größer das Momentum desto schneller Bewegt sich der Gradient abwärts. Da am Anfang eines Lernphase der gradient überlicherweise recht hoch ist empfiehlt sich zunächst mit einen niedrigen Momentum zu Arbeiten da sonst die Gefahr besteht über das globale Optimum hinüberzuschießen. Wenn

nun das Training stagniert welches auf Gründe der Aufbau der Loss-Funktion zurückzuführen ist das zur Nähe des globalen Optimums flache Täler entstehen welche das Trainings verlangsamen. und es zu keiner Verbesserung kommt kann man durch momentum erzwingen größere Gradienten sprünge einzugehen und sich somit schneller zu einem globalen Optimum zu bewegen oder aber aus einem lokalen Optimum hinaus Richtung eines globalen Optimum(? , ?). .

Here's a popular story about momentum gradient descent is a man walking down a hill. He follows the steepest path downwards; his progress is slow, but steady. Momentum is a heavy ball rolling down the same hill. The added inertia acts both as a smoother and an accelerator, dampening oscillations and causing us to barrel through narrow valleys, small humps and local minima. This standard story isn't wrong, but it fails to explain many important behaviors of momentum. In fact, momentum can be understood far more precisely if we study it on the right model. One nice model is the convex quadratic. This model is rich enough to reproduce momentum's local dynamics in real problems, and yet simple enough to be understood in closed form. This balance gives us powerful traction for understanding this algorithm(? , ?).

For a step-size small enough, gradient descent makes a monotonic improvement at every iteration. It always converges, albeit to a local minimum. And under a few weak curvature conditions it can even get there at an exponential rate.

<https://distill.pub/2017/momentum/>

2.1.3 Batch Normalisation Wie in Kapitel backpropagation Algorithmen gezeigt wurde wird durch den stochastischen Gradienten Vorteile gegenüber des normalen Gradienten Verfahren ergeben. Durch das der Input in jeden Layer abhängig von den vorherigen Layer abhängig. Dadurch können Änderungen in vorherigen Layer große Auswirkungen in tieferen Layer im Netzwerk haben. Dadurch resultiert dann das in Trainingsläufen die Verteilung von jedem Layer Input sich während des Trainings verändert die verlangsamt das Training erheblich (? , ?). Dieses Problem wird als internal covariate shift definiert. Wenn sich die Input Verteilung von einem lernenden System verändert macht es einen covariate shift durch (? , ?). Um dies zu verhindern zeigten Sergey Ioffe und Christian Szegedy (? , ?) eine Methode welche Batchnormalization genannt wird. Je mehr Layer das Netzwerk hat desto stärker wird der internal covariate shift. Batch normalization besteht aus zwei Algorithmen. Algorithmus 1 verändert den ei-

gentlichen Input von Layer n zu einen normalisierten Input y und Algorithmus 2 verändert das eigentliche Training eine batch normalisierten Netzwerkes.

Algorithm 1: Batch Normalisierung angewand auf x über Input bei Mini-Batch

```

1 Input: Werte von x über einen Mini-Batch  $B = \{x_1, \dots, x_n\}$ 
2  $\mu_B \leftarrow \frac{1}{m} \sum_{i=1}^m x_i$ 
3  $\sigma_B \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2$ 
4  $\hat{x}_{iB} \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B + \epsilon}}$ 
5  $y_i \leftarrow \gamma \hat{x}_i + \beta \equiv BN_{\gamma;\beta}(x_i)$ 
6 Output:  $\{y_i = BN_{\gamma;\beta}(x_i)\}$ 

```

In Schritt 2 des Algorithmus 1 wird der Erwartungswert für alle Input von Mini-Batch B berechnet. In Schritt 3 die Varianze. In Schritt 3 wird nun der der normalisierte x_i berechnet welche dann mit dem β und γ multipliziert werden. Diese Werte sind neue gewichte im Neuronalen Netzwerk welche während des Trainingsporcesses angepasst werden. ϵ in der Gleichung in Zeile 4 ist nur dafür da damit nicht durch 0 geteilt werden kann. In Zeile 8 - 11 werden die Inferenz schritte beschrieben bei den der Minibatch des Trainings ersetzt wird.

Algorithm 2: Training mit Batch-Normalisierungs Netzwerk

```

1 Input: Netzwerk N mit trainierbaren Parameter  $\theta; \{x^{(k)}\}_{k=1}^n$  bis
2 Ntr BN  $\leftarrow$  N
3 a
4 a
5 a
6 a
7 a
8 a
9 a
10 a
11 a
12 a

```

Schritt 1 bis 5 des Algorithmus 2 baut eigentlich nur das neuronales Netzwerk durch die transformationen aus Algorithmus 1 um. In Schritt 6 und 7 geht es darum die Parameter γ und β zu trainieren. Dieses passiert mit den üblichen Backpropagation Algorithmus wären des Allgemeinen Training des Netzwerkes.

Je mehr Layer das Netzwerk hat desto stärker wird der internal covariate shift

Neuronale Netzwerke sind universelle Funktions approximierer (?, ?). Es sei erfohrzuheben das zwar Feedforward Neuronale Netzwerke mit hidden Layer unisversel sind. Aber nicht jede aktivierungsfunktion ist für alle probleme gleich effizient. (?, ?). In Kapitel BLABLA wurde gezeigt das Neuronale Netzwerke

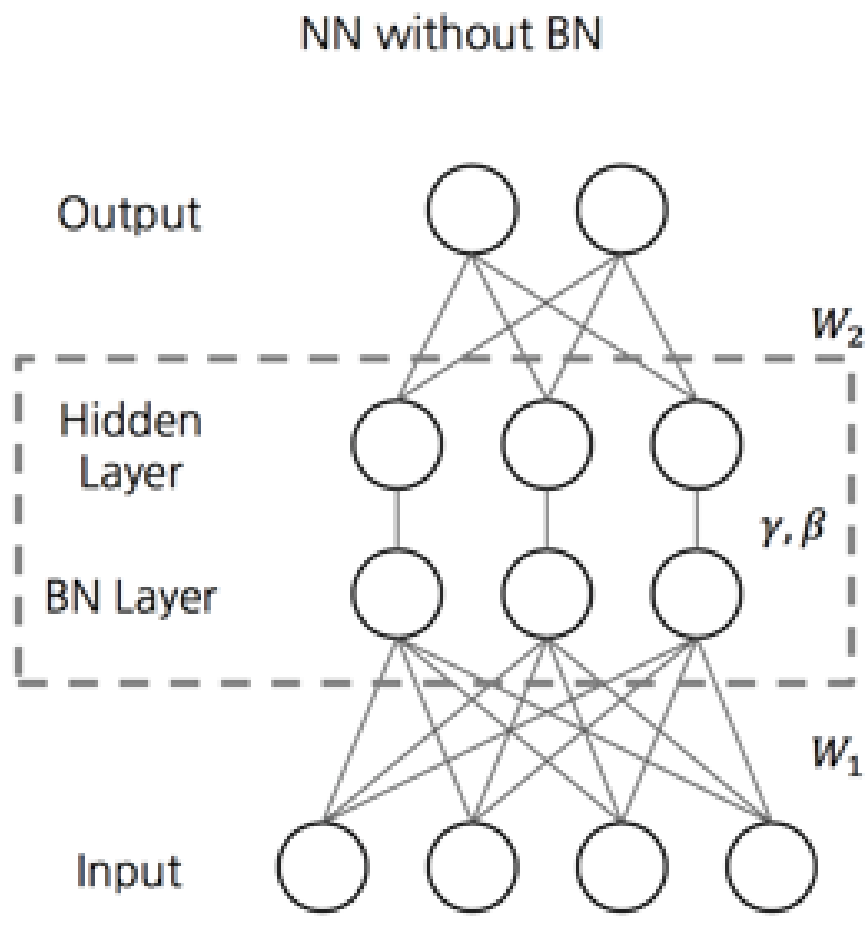


Abb. 7. Neuronales Netzwerk mit Batch Normalization Layer

durch Matrizen dargestellt werden können und das durch simple Matrixmultiplikation der Output von Layern berechnet werden können. Die Aktivierungsfunktion sorgen dafür das das ANN auch nicht lineare Funktionen erlernen kann. Also das unser Input X und unser Output Y immer proportional als $Y = X \cdot k$. Wobei k eine Konstante ist. Wie wir in Kapitel BLABLA gesehen haben möchten wir das unser Input linear trennbar ist damit wir unsere Input in Klassen einteilen können. Würde X nun aber nicht linear trennbar sein wie Beispielsweise in Abbildung UNTEN gezeigt könnten wir durch die lineare Transformation des Inputs keine Trennung von X erreichen. <http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/> Mit den einführen von nicht linearen Funktionen können der Raum der durch die Layer dargestellt wird gedreht gewendet und gezogen werden. Aber es scheitert, zerbricht oder faltet in. Mit den einführen von nicht linearen Funktionen können der Raum der durch die Layer dargestellt wird gedreht gewendet und gezogen werden.

Ein Neuron hat einen Input x_n sowie ein Gewicht w_{ij} und einen Bias b . der Output eines Neurons ist definiert durch $y = x_n \cdot w_{ij} + b$

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

$$\zeta(x) = \log(1 + \exp(x))$$

2.1.4 Ziel-Funktion Die Ziel-Funktion muss Differenzierbar sein. Das heißt unsere Funktion $f: D \rightarrow R$ ist differenzierbar an der Stelle x_0 an der Stelle. Wenn nun $f': x \rightarrow f'(x)$ an jeden Punkt x^n ableitbar ist, ist f differenzierbar. Die Aufgabe der Ziel-Funktion ist es zu messen wie gut unsere Modell θ $f^*(x)$ approximiert. Man verwendet auch gerne den Begriff Kosten, also wieviel Kosten unser Modell erzeugt. Daher wird die Ziel-Funktion auch Kosten-Funktion genannt. Die Wahl welche Funktion gewählt wird ergibt sich aus der Aufgabe des ANN soll es zur Regression Diese kann für Klassifikation und Regression Aufgaben hergenommen werden. Bei Regression soll eine kontinuierlicher Variablen von θ als Output generiert werden wohin gegen bei Klassifikationsproblemen der Output Klassen-Labels darstellen. Es gibt verschiedene Ziel Funktionen im folgenden wird die Categorical Cross Entropy Loss Function vorgestellt (?, ?). Diese wird verwendet wenn man das Ziel hat ein Klassifikationsproblem zu Lösen und $n \geq 1$ Klassen hat. Diese ist definiert als:

$$\hat{q}(c | x) = \arg \min_{q(c|x)} \left\{ - \sum_n \log q(c_n | x_n) \right\}$$

Wobei $x_n: n=1, \dots, N$ die Trainingsdaten sind ist und $c_n: n=1, \dots, N$ die mögliche Klassen.

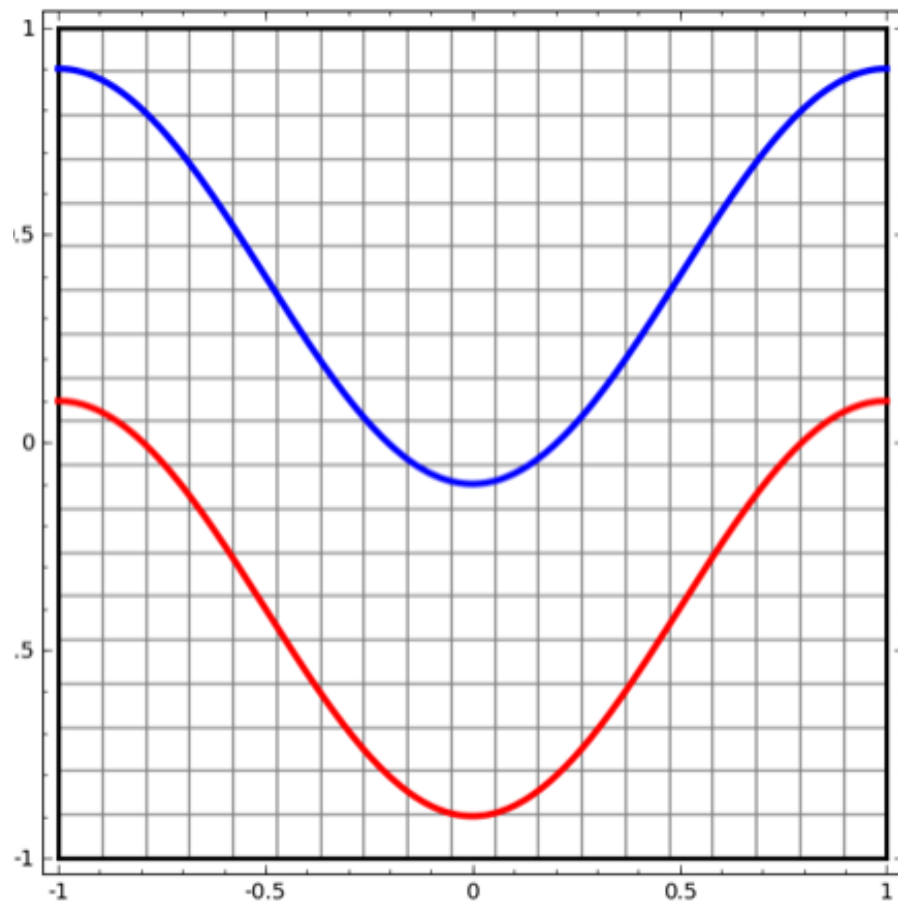


Abb. 8. Nicht trennbar

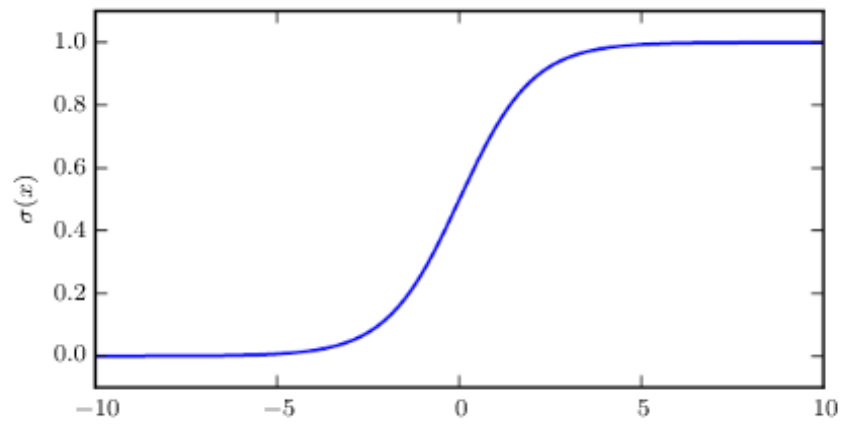


Abb. 9. Sigmoid Funktion

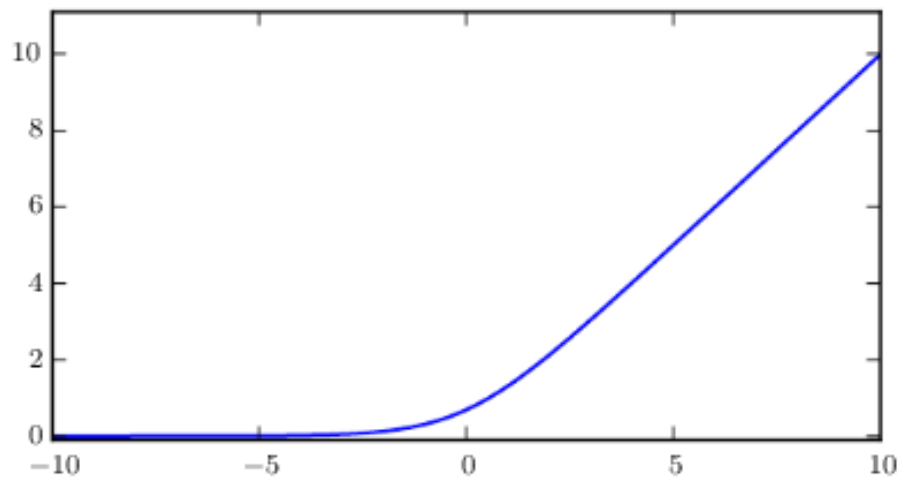


Abb. 10. Softmax

2.2 Datenformat

Punktwolken sind eine Menge von N Punkten welche im Vektorraum dargestellt werden können. Jeder Punkt n Element von N wird durch seine (x,y,z) im Euklidischen Raum dargestellt. Punkte können zusätzliche Features gegeben werden wie Farbe oder Material. Es gibt unterschiedliche Dateiformate welche für die Abspeicherung von Punktwolken herangezogen werden können Beispiele dafür sind PLY, STL oder OBJ. Das Polygon File Format (PLY) speichert die einzelnen Koordinaten in einer Liste diese wird Vertex List genannt. In Abb. 13 kann eine Beispieldatei entnommen werden. Diese kann als Menge betrachtet werden die jeweiligen Punkte sind zwar geordnet in dieser Liste gespeichert jedoch spielt es keine Rolle für den Punktwolkencompiler bei der Visualisierung der Liste an welcher Listenposition ein jeweiliger Punkt geführt wird, die Liste wird jedes mal gleich angezeigt egal welche Permutation der einzelnen Punkte in der Liste durchgeführt wird. In Abb. 11 kann eine visualisierte Punktwolke eines Tabakblattes entnommen werden.

2.2.1 Woher stammen die Daten Die Daten stammen von TERRA-REF Feld Scanner von der University von Arizona Maricopa Agricultural Center and USD Arid Land Research Station in Maricopa. Es ist der größte Feld crop scanner. In Abb. 13 ist dieser abgebildet. In Abb. 14 ist der Scankopf des TERRA-REF dargestellt mit den unterschiedlichen Scanmöglichkeiten für das zu scannende Objekt.

Der 3D Laser Scanner erzeugt 3D Punktwolken. Dabei werden die Objekte durch den Scanner erfasst und eine 3D Repräsentation welche durch Punkte in einen 3 Dimensionalen Koordinaten System erfasst werden können dargestellt. Dabei wird ein Laser über das zu scannende Objekt gefahren durch Reflexion des Laserstrahls auf der Oberfläche des Objektes können x,y,z Koordinaten des jeweiligen Punktes auf den Objekt bestimmt werden. Durch die Sammlung einzelner Scanpunkte entsteht eine 3D-Repräsentation eines Gegenstandes.

2.2.2 Das Problem mit 3D-Data bei Machine Learning Ansätzen Vergleicht man 3D-Data auf ihre Dimensionalität mit anderen Datenformaten wie Bild,- , Audio ,- Textdateien. Wie im gezeigten Kapitel "Woher stammen die Daten" gezeigt sind 3D-Daten als Menge gespeichert werden in denen keine Relation untereinander besteht das heißt es ist für den Punktwolkencompiler egal auf welchen Platz die einzelnen Punkte abgespeichert werden angezeigte Punktwolke schaut gleich aus. Vergleichen wir nur ein Bild mit 512 Pixeln mit RGB Farbkämen sind wir bei einer Dimension von 391680 bei 3 Farbkanälen zwischen 0-255. Vergleichen wir nun das mit einer Punktwolke in einen 125 cm³ großen Bereich. Da die einzelnen Koordinaten eines Punktes als Rationale Zahlen dargestellt wird und rationale Zahlen abzählbar Unendlich ist ist unser Suchraum unendlich Zeichen "größer. Dies führt zu einem erheblichen Mehraufwand für Machine Learning Ansätzen wie Deep Learning für Punktwolken.

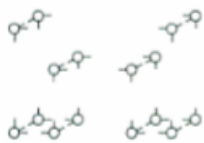
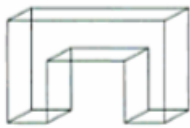
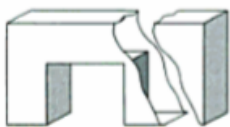

Dimension der Elemente	Element	Modelltyp
0D	<p>Punkt</p> 	Eckenmodell
1D	<p>Linie</p> 	Kantenmodell
2D	<p>Fläche</p> 	Flächenmodell
3D	<p>Volumen</p> 	Volumenmodell (Körpermodell)

Abbildung 1: CAD-Elemente und Modelltypen (Friedrich, 2012, S. 27)

Abb. 11. Punktwolke eines Tabakblattes


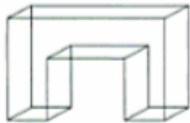
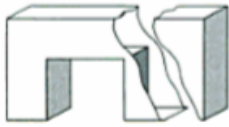

Dimension der Elemente	Element	Modelltyp
0D	Punkt 	Eckenmodell
1D	Linie 	Kantenmodell
2D	Fläche 	Flächenmodell
3D	Volumen 	Volumenmodell (Körpermodell)

Abbildung 1: CAD-Elemente und Modelltypen (Friedrich, 2012, S. 27)

Abb. 12. Polygon File Format

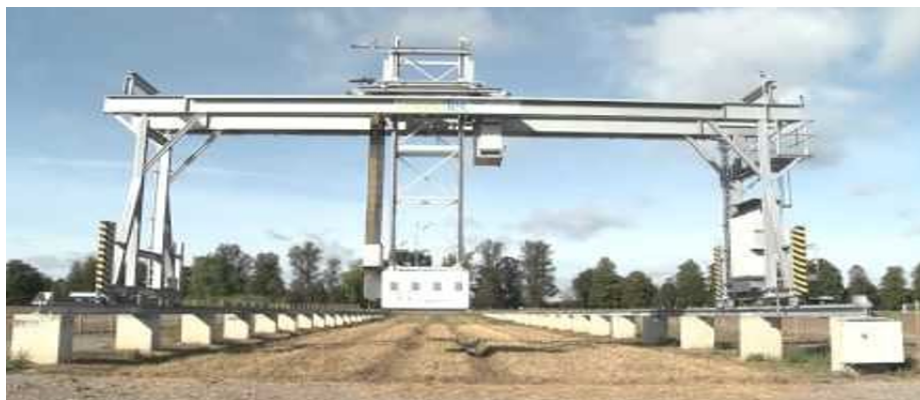


Abb. 13. TERRA-REF Feld Scanner

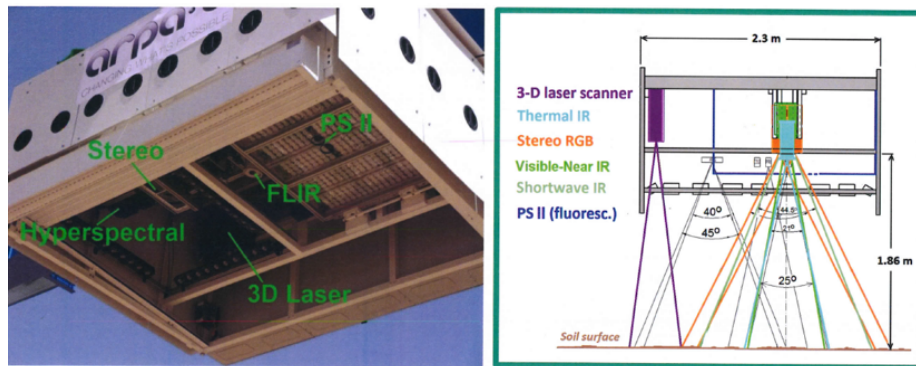


Abb. 14. Scankopf

Ein weiteres Problem für Deep Learning ist das die aus Kapitel Convolutional Neural Network beschriebenen Convolutional Layer einen großen Beitrag bei den Fortschritt von Deep Learning gebracht hat. Da sie helfen die Strukturen von strukturierten Daten zu lernen und den latenten Raum zu entdecken. Da jedoch Pointclouds unstrukturiert sind hilft es nicht diese Tools bei PC einzusetzen und bringen keinen mehr gewinn.

2.3 Deep Learning und Informationstheorie

Von Deep Learning wird gesprochen wird ANN mehr als 1 Layer besitzen und dadurch Komplexere Aufgaben lösen können. Der Trainingsalgorithmus ist immernoch mit den Backproagation Algorithmus. Um das Konzept von Deep Larning besser zu verstehen kann man diese aus den Informationstheoretischen Blickwinkel betrachten. Tischby and Schwarz-Ziv (?, ?) untersuchten dabei Neuronale Netzwerke in Verbindung mit Deep Learning.

Das ins 18 dargestellte Netzwerk kann als Markov Kette betrachtet werden. Wobei jeder Layer $h^n:1,...,N$ als ein Zustand einer Markov Kette betrachtet werden kann. Wobei der Übergang durch von Informations als $h_i \rightarrow h_{i+1}$ dargestellt werden kann. Da es keine Rekursion gibt und die Information von Input Layer X durch h_{arg} zu Output Layer Y durch fließt. Dabei untersuchten sie wie sich die Transinformation $I(X;Y)$ von Input Variable X zu Output Variable Y verhält gegeben durch die Multivariate Verteilung $p(X,Y)$. Dieses Prinzip wird Information Bottleneck Prinzipale genannt (?, ?).

2.4 Convolution Neural Networks

einbauen wie bild bearbeitet ist: <https://arxiv.org/pdf/1801.00634.pdf>

Convolution Neural Networks(CNN) sind eine besondere Art von künstlichen neuronalen Netzwerken, sie sind dafür konzipiert auf Datensätzen zu arbeiten welche in eine Matrix Form gebracht worden sind. Der Input eines CNN können

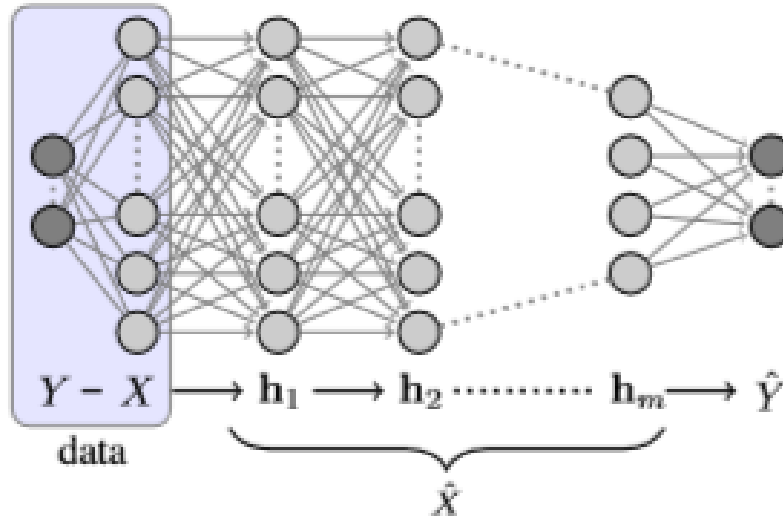


Abb. 15. Neuronal Network als Markov Chain

beispielsweise Bilder sein welche durch die Matrix $A = \begin{bmatrix} a_1 & a_1 & \dots & a_n \\ b_1 & b_2 & \dots & b_n \\ \vdots & n_n & \ddots & \vdots \\ x_1 & x_2 & x_3 & x_n \end{bmatrix}$

dargestellt werden. Jedes Element x_{ij} stellt einen Pixel eines Bildes da, wobei $x_n \in [0, 255]$. Die Matrix $A^{w \cdot b \cdot c}$ stellt $w \cdot b \cdot c = N$ dimensionale Matrix da. Wobei w die Länge und b Breite des Bildes entspricht. c sind die Farbspektren eines Bildes und sind in einen RGB-Farbraum 3 beziehungsweise in einen schwarz-weiß Bild 1. Nachdem der Input eines CNN definiert ist kommt nun der Aufbau. CNN setzen sich aus mehrere Schichten von Convolution Layern zusammen. Ein Netzwerk kann mehrere N-Layer haben. Wobei jeder Layer aus mehreren Convolution oder auch Kernels genannt, zusammengesetzt ist. Ein Aufbau kann aus Abbildung 34 entnommen werden(?, ?).

Die Kernels, also die einzelnen Filter, von den jeder der N-Layer k besitzt sind $K^{n \cdot n}$ Matrizen jedes k_{ij} in einem Filter entspricht einen aus der üblichen Neuronalen Netzwerk Architektur bekannten Gewichte. Diese Gewichte werden dann durch den Backpropagation-Algorithmus in der Trainingsphase des Netzwerkes angepasst um den Verlust der Loss-Funktion durch bestimmten des Gradienten zu minimieren. Das durch die Abbildung 34 dargestellte Subsampling ist der Output aus den Convolutional Layern(?, ?).

Da Input und Kernel unterschiedliche Größen haben und man den gesamten Input mit den Kernel abdecken möchte, bewegt sich der Filter um s Position

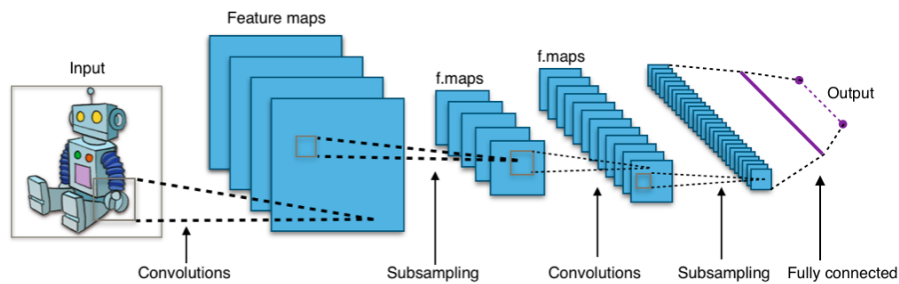


Abb. 16. Convolutional Neural Network

auf den Input und führt erneut einen Berechnungsschritt durch. Dieser Vorgang wird Stride genannt. An jeder Position wird das Produkt von jedem x_{ij} des Input und k_{ij} des Kernel durchgeführt. Anschließend werden alle Produkte aufsummiert. In Abbildung 17 ist dieser Vorgang verdeutlicht. Zusätzlich gibt es die Möglichkeit für das sogenannte Zero Padding P. Dabei werden mehrere 0 um die Input Feature Map, am Anfang und Ende der Axen anfügt. Dies ist notwendig wenn Kernel und Input Größe nicht kompatibel zueinander sind. Die Anzahl der möglichen Positionen ergeben sich aus Kernel Größe und den Input des jeweiligen Kernel sowie des Strides. Die Output Größ W kann berechnet werden durch $W = (W-F+2P)/s+1$. Wobei F für die Größe des Kernel steht (?, ?).

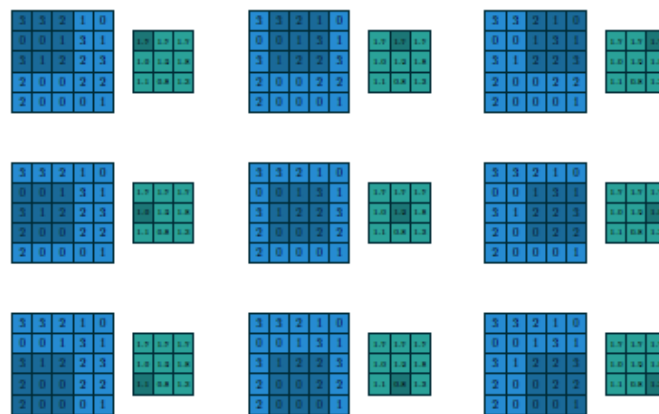


Abb. 17. Convolution Beispiel

Um besser zu verstehen welche Auswirkungen die Anzahl der Kernels in Layer n auf die Größe des Outputs von n und die Anzahl der Kernels in Layer n+1 für den nächsten Layer haben, wird ein Beispiel aufgezeigt. Der erste Layer

hat 20 Kernels mit der Größe 7×7 und Stride 1. Der Input A für einen Kernel K ist ein 28×28 Matrix. Der Output aus diesen Filter sind 20 22×22 Feature Maps. Würde der Input ein $28 \times 28 \times 3$ Bild mit 3 RGB Channels sein, der Output 60 22×22 Feature Maps. Allgemein kann Convolution Layer als Supersampling gesehen werden und Stride gibt an wieviele Dimensionen bei diesen Prozess pro Convolution Layer entfernt werden soll. Der letzte Layer ist ein fully-connected Layer welcher den typischen Anforderungen von ANN entspricht (?, ?).

Transposed Convolution, auch genannt Fractionally Strided Convolution oder Deconvolution ist eine Umkehrfunktion von der üblichen Convolution. Es verwendet die gleichen Variablen wie Convolution. Dabei wird ein Kernel K mit der Größe $N \times N$ definiert der Input I mit der Größe $N \times N$ und Stride $s = 1$. Deconvolution kann wie Convolution angesehen werden mit Zero Padding auf dem Input. Das in Abbildung 18 gezeigte Beispiel zeigt einen deconvolution Vorgang mit eine 3×3 Kernel über einen 4×4 Input. Dies ist gleich mit einen Convolution Schritt mit einen 3×3 kernel auf einen 2×2 Input und einer 2×2 Zero Padding Grenze. Convolution ist Supersampling und mit Deconvolution wird Upsampling betrieben. Durch diesen Schritt kommt es zu einer Dimensionserhöhung des Inputs. Die Gewichte der Kernels bestimmen wie der Input transformiert wird. Durch mehrere Schichten von Deconvolution Layer kann von einer Input Größe $N \times N$ auf eine Output Größe $K \times K$, wobei $K > N$ mit Abhängigkeit von Kernel und Stride abgebildet werden(?, ?).

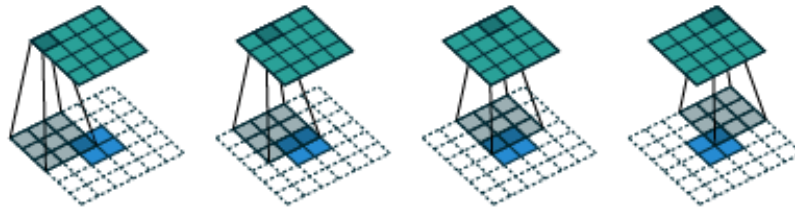


Abb. 18. Deconvolution Beispiel

2.5 Autoencoder

Autoencoder sind ein andere Art von Modell aus den Bereich der generativen Modelle. Ihre Aufgabe besteht darin einen Input zu komprimieren und aus der komprimierten Information den Input wieder herzustellen. Die Aufgabe kann auch als Rekonstruktion benannt werden. Die Technik auf welche Autoencoder zurückgreifen nennt sich Dimensionreduction. Dabei werden die Dimension der Daten so reduziert und Informationen bei zu behalten welche als relevant gelten. Diese Technik finden auch in anderen Machine Learning Anwendung wie in beispielsweise der Principale component anaysis(PCA) anwendung. Ein Autoencoder besteht aus 2 Bestandteilen. Erstens ein Encoder e parametrisiert mit ϕ welcher einen Input $x \in \mathbb{R}^i$ wo x ein Vektor der länge i ist und damit die Input Dimension bestimmt. Dieser wird durch den Encoder auf einen Vektor z^k abgebildet wobei $k < i$ ist. Zweitens der Decoder d parametrisiert durch θ bekommt als Input z^k und mappt z auf x^l wobei $l = i$. Und somit die gleiche Dimension wie der Input. Aufgabe ist es nun das der Encoder den Input z so gut komprimiert das der Decoder es schafft das $x \approx z$. Eine grafische Darstellung kann aus Abb. 26 entnommen werden.

Die Paramter ϕ und θ werden können durch Deep Learning erlernt werden. Und können beispielsweise durch fully-Connected Layer,Convolutional-Layer oder Deconvolutional Layer modelliert werden. Eine Metrik um zu messen wie gut das Modell seine Aufgabe erfüllt könnte Beispielsweiße Cross-Entropy functionen sein welche schon in Kapitel vorgestellt worden sind. Weitere spezifische Zielfunktionen für Autoencoder welche mit 3D-Daten arbeiten werden unter Kapitel vorgestellt.

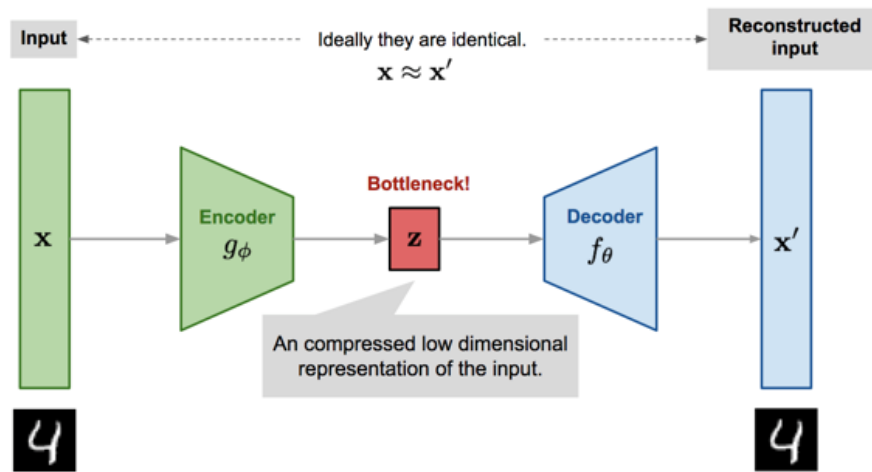


Abb. 19. autoencoder

2.5.1 Zielfunktion 3D-Punktwolken Autoencoder Da in der Arbeit der Input 3d Pointcloud sind und diese Sets und diese invariant zu ihren Permutationen. Das heißt ändere ich die Anordnung meiner einzelnen Punkte in meinen Set bleibt das Ergebnis unverändert. sind sind kann auf übliche Zielfunktionen nicht zurückgegriffen werden da diese mit strukturierten Input Daten arbeiten. Das Problem dabei besteht wenn ich 2 unterschiedliche Sets von Punkten habe wie messe ich um heraus zu finden wie hoch die Discrepanz zwischen den beiden Sets ist. <http://graphics.stanford.edu/courses/cs468-17-spring/LectureSlides/L14>

Generative Modelle haben das Ziel eine Wahrscheinlichkeitsverteilung zu erlernen. Anschließend kann diese als ein Modell genutzt werden und Samples zu erzeugen. Die Modelle können dabei beispielsweise auf ANN oder Markov Chains trainiert werden(?, ?). Im folgenden liegt der Fokus auf ANNs. Ein mögliches Anwendungsbeispiel wäre es dem generativen Modell, Bilder von bestimmten Objekten zunächst als Trainingsdaten zu geben. Anschließend können von der erlernten hochdimensionalen Wahrscheinlichkeitsverteilung Samples gezogen werden und neue Bilder erzeugt werden, welche nicht im Trainingsdatensatz vorhanden waren. Allgemein gehalten können jegliche Typen von Daten wie Text, Bild oder Audiodateien für generative Modelle herangezogen werden. Es gibt unterschiedliche Typen von generativen Modellen, welche sich vom Aufbau des neuronalen Netzwerks und der Zielfunktion unterscheiden. Beispiele dafür sind Boltzmann Maschine, Autoencoder oder Deep Belief Networks(?, ?). Diese Arbeit beschäftigt sich mit einem anderen Vertreter, dem Generativ Adversarial Network(GAN). In den letzten Jahren konnte sich das GAN als best practice Ansatz bei den generativen Modellen herausarbeiten was Performancegründe bei der Trainierbarkeit und Qualität der generierten Daten zu Grunde liegt(?, ?). Die Modelle arbeiten nach der Maximum Likelihood Schätzverfahren(ML-Schätzer) in dem die Parameter θ dahingegen angepasst werden, dass die unsere beobachteten Daten am besten passen. Man kann ML-Schätzer als Kullback-Leibler(KL) Divergenz darstellen und das generative Modell das Ziel haben die KL Divergenz zwischen den Trainingsdaten P_r und den generierten Daten P_g zu minimieren. In Abbildung 22 ist dieser Prozess dargestellt. Diese Modelle versuchen dann die diese Cost Funktion

$$KL(P_r||P_g) = \int_x P_r \log \frac{P_r}{P_g} dx$$

zu minimieren. Wenn nun beide Verteilungen $P_r = P_g$ sind, hat das Modell sein Minimum Loss erreicht und θ muss nicht mehr angepasst werden. Interessant wird es, wenn $P_r \neq P_g$. Wenn $P_r > P_g$ führt, dass dazu dass das Integral schnell gegen unendlich konvergiert. Was dazu führt, dass hohe Kosten entstehen wenn die Verteilung von generativem Modell erzeugt, nicht die Daten abdeckt. Wenn nun $P_r < P_g$ ist, dann bedeutet das, dass x eine niedrige Wahrscheinlichkeit hat aus unseren Trainingsdaten zu kommen aber eine hohe Wahrscheinlichkeit von den Generator erzeugt zu werden. Dann würde sich die KL gegen 0 konvergieren. Was zur Folge hat, dass der Generator falsch ausschauende Daten generiert aber keine Kosten dafür erzeugt werden und im Umkehrschluss es zu keinen

Veränderungen unseren Θ kommt. Man vermutet das dies der Grund für die Problematiken von generativen Modellen wie Autoencoder und CO sind. Dies ist aber noch kein abgeschlossenes Problem und wird weiterhin erforscht(?). Unter GAN werden wir auf dieses Problem erneut aufgreifen und es wird gezeigt inwiefern sich GAN dieses Problem angeht. Generative Modelle gehören zu einem Bereich des unüberwachten Lernens, da keine Labels für die Trainingsdaten gebraucht werden. Probleme welche diese Modelle haben sind beispielsweise, dass Autoencoder zwar mit wenig Trainingsaufwand trainiert werden können jedoch sind die generierten Bilder sehr trüb. Allgemein haben Autoencoder und Co. Vorteile im Lernen des latenten Raums von Objektklassen, weisen aber Probleme beim Generieren von neuen Daten auf. Da gezeigt wurde das im Deep Learning Bereich die discriminativen Modelle mit Zunahme der Daten stark an Leistung zunehmen. Und GANs Stärke in der Datengeneration in guter Qualität liegt. Kann es seine Vorteil gegenüber den anderen Modellen ausspielen(?).

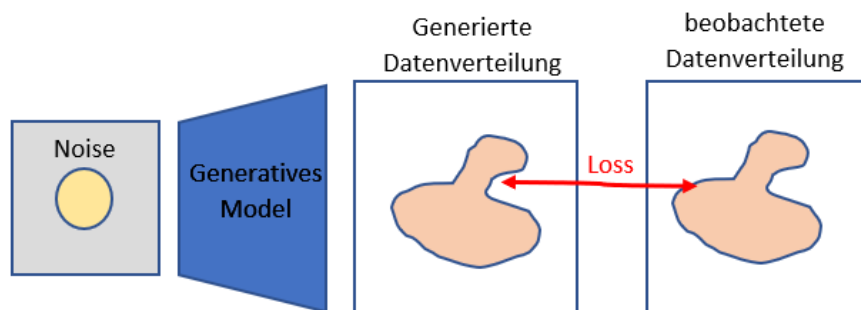


Abb. 20. Generative Modelle

Eine weitere Möglichkeit speziell für Autoencoder für Punktwolken ist die Earth Mover distance(EMD). Sind X^1 und X^2 zwei Punktwolken mit jeweils x^n definierten Punkten.

$$d^{\text{EMD}} = \min_{\theta: X_1 \rightarrow X_2} \sum_{x \in X_1} \|x - \theta(x)\| \text{ wobei } \theta: X_1 \rightarrow X_2 \text{ bijektiv ist}$$

bei den Ergebnissen von ZITAT 3D-GAN hat sich eine Zielfunktion besser herauskristalisiert die Chamfer distance(CD). Wie auch zuvor sind X^1 und X^2 zwei Punktwolken mit jeweils x^n definierten Punkten.

$$d^{\text{CD}}(X_1, X_2) = \sum_{x \in X_1} \min_{y \in X_2} \|x - y\| + \sum_{y \in X_2} \min_{x \in X_1} \|x - y\|$$

Der unterschied ergibt sich zwischen den beiden das bei der EMD von der Ausgangswolke die Punkte zu der anderen jeweils optimiert werden. Wohingegen bei der CD die distanzen von und zu der Ausgangspunktwolke berechnet werden. Dies geht sehr gut aus den Grafiken 1 und 2 hinaus in welche die distanceberechnungen der einzelnen Punkte dargestellt wird.

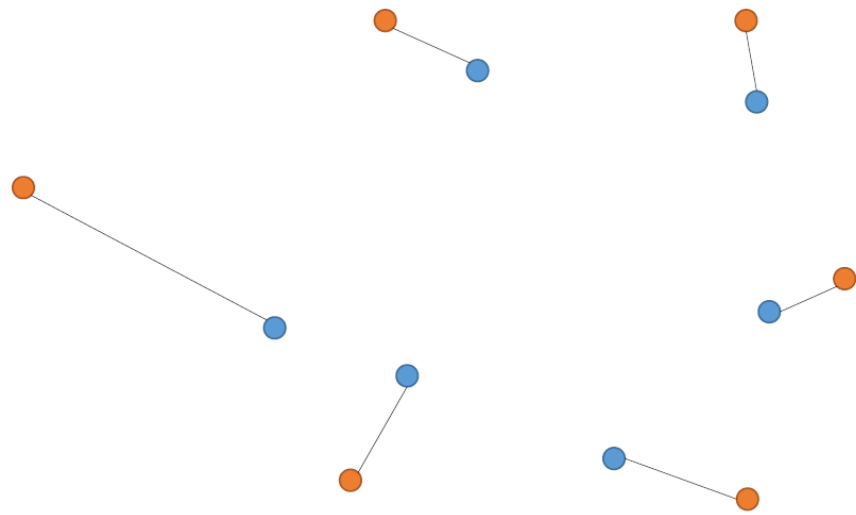


Abb. 21. Generative Modelle

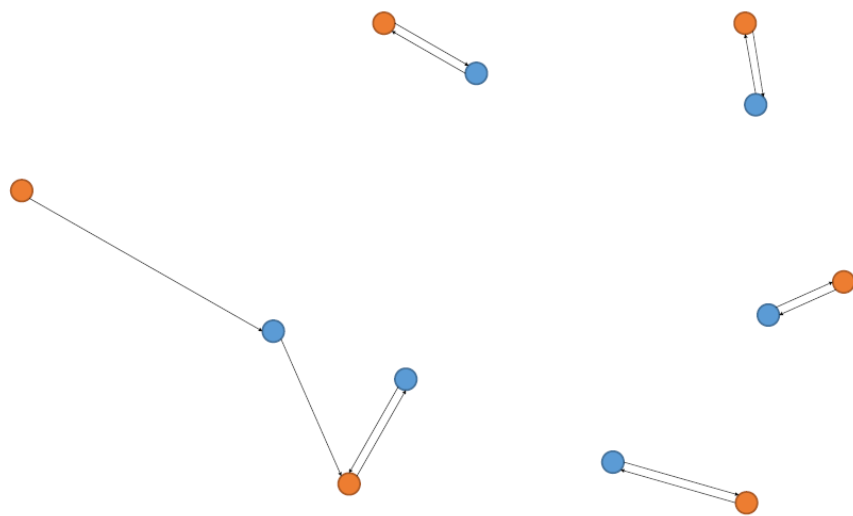


Abb. 22. Generative Modelle

2.6 Generative Adversarial Network

Ein GAN besteht aus zwei ANN, dem Discriminator D und dem Generator G. Das Ziel des G ist es, Daten x zu erzeugen, welche nicht von Trainingsdaten y unterschieden werden können. Dabei wird eine vorangegangene Input Noise Variable $p_z(z)$ verwendet, welche eine Abbildung zum Datenraum $G(z; \Phi_g)$ herstellt. Dabei sind Φ_g die Gewichte des neuronalen Netzwerkes von G. Der Discriminator hat die Aufgabe zu unterscheiden, ob der jeweilige Datensatz von G erzeugt wurde und somit ein fake Datensatz ist, oder von Trainingsdaten y stammt(?). Die Zusammensetzung zwischen den beiden Netzwerken kann aus Abbildung 23 entnommen werden.

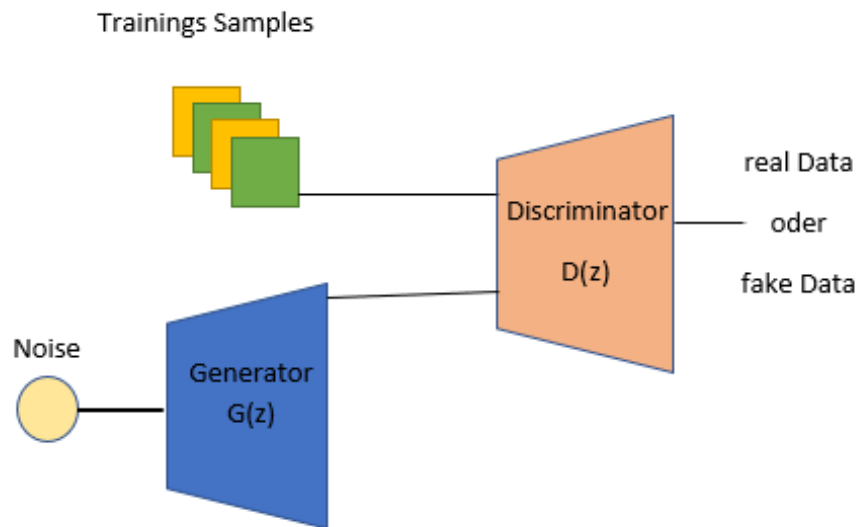


Abb. 23. Generativ Adversarial Network

Der Discriminator ist definiert durch $D(x; \Phi_d)$. Wobei Φ_d die Gewichte des Discriminators sind und $D(x)$ die Wahrscheinlichkeit ist, dass x von den Trainingsdaten stammt und nicht von p_g . Die Wahrscheinlichkeitsverteilung für unsere Trainingsdaten ist p_r . Im Training werden dann Φ_d so angepasst, dass die Wahrscheinlichkeit Trainingsbeispiele richtig zu klassifizieren maximiert wird. Und Φ_g wird dahingehen trainiert die Wahrscheinlichkeit zu minimieren, so dass D erkennt dass Trainingsdatensatz x von G erzeugt wurde. Mathematisch ausgedrückt durch $\log(1 - D(G(z)))$. Die gesamte Loss-Funktion des vanilla GAN

ist definiert als

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

diese beschreibt ein Minmax Spiel zwischen G und D. Welches das globale Optimum erreicht hat wenn $p_g = p_r$. Das heißt, wenn die Datenverteilung, welche von G erzeugt wird, gleich der unserer Trainingsdaten ist(?, ?). Das Training erfolgt durch den folgenden Algorithmus:

Algorithm 3: Minibatch stochastic gradient descent Training für Generative Adversarial Networks. Die Anzahl der Schritte welche auf den Discriminator angewendet wird ist k

```

1 for Anzahl von Training Iterationen do
2   for k Schritte do
3     • Sample minibatch von m noise Samples  $z^{(1)}, \dots, z^{(m)}$  von noise
       $p_g(z)$ 
4     • Sample minibatch von m Beispielen  $x^{(1)}, \dots, x^{(m)}$  von Daten
      Generationsverteilung  $p_{\text{data}}(x)$ 
5     • Update den Discriminator zum aufsteigenden stochastischen
      Gradienten:
6      $\nabla_{\Phi_d} \frac{1}{m} \sum_{i=1}^m [\log D(x^{(i)}) + \log(1 - D(G(z^{(i)})))]$ 
7   end
8   • Sample minibatch von m noise Samples  $z^{(1)}, \dots, z^{(m)}$  von noise  $p_g(z)$ 
9   • Update den Generator mit den absteigenden stochastischen
      Gradienten:
10   $\nabla_{\Phi_g} \frac{1}{m} \sum_{i=1}^m \log(1 - D(G(z^{(i)})))$ 
11 end
```

Beim Training wird ein stochastischer Minibatch von mehreren Trainingsdaten gleichzeitig erstellt. Dies soll dabei helfen, dass der Generator sich nicht auf bestimmte Gewichte fest fährt und auf Trainingssätze kollabiert. So weisen die erzeugten Daten mehr Variationen auf (?, ?). D wird zunächst in einer inneren Schleife auf n Trainingsätzen trainiert, womit man Overfitting von D vermeiden will, was zur Folge hätte, dass D nur den Trainingsdatensatz kopieren würde. Deshalb wird k mal D optimiert und ein mal G in der äußeren Schleife.

Ein möglicher Aufbau von GAN wird in Abbildung 24 dargestellt. Dies ist das sogenannte Deep Convolution GAN(DC GAN), welches dafür konzipiert wurde auf Bilddaten zu arbeiten. Dabei besteht der Generator aus mehreren Schichten von Deconvolution Layern. Welche den Input Noise Variable $p_z(z)$ auf y abbildet. D besteht aus mehreren Schichten von Convolution Layern und bekommt als Input die Trainingsdaten, oder die von G erzeugten Y , und entscheidet über die Klassifikation($?, ?$).

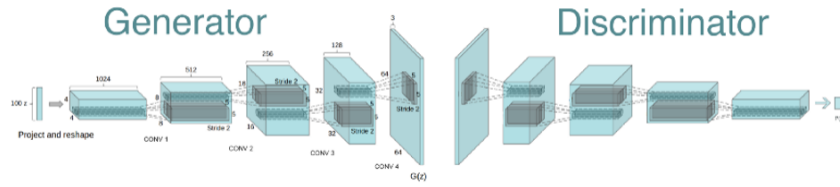


Abb. 24. Deep Convolutional GAN

Wie unter Generativen Modellen gezeigt wurde kann das asymmetrische Verhalten der KV Divergenz zu schlechten Trainingsergebnissen führen. Goodfellow ($?, ?$) zeigte, dass sich die MinMax Loss-Funktion des GAN auch als Jensen-Shannon Divergenz(JS Divergenz) darstellen lässt. Diese ist definiert als

$$D_{JS}(P_r||P_g) = \frac{1}{2}D_{KL}(P_r||\frac{P_g+P_r}{2}) + D_{KL}(P_g||\frac{P_g+P_r}{2})$$

wobei P_r die Wahrscheinlichkeitsverteilung der Trainingsdaten ist und P_g die des Generators. Huzár ($?, ?$) zeigte, dass durch das symmetrische Verhalten der JS Divergenz ein potentiell besseres Trainingsergebnis entstehen kann, im Vergleich zu der KL Divergenz. Damit zeigte weshalb GANs im Vorteil gegenüber anderen generativen Modellen sind. Abbildung 25 veranschaulicht dieses Konzept. Der linke Graph zeigt 2 Normal Verteilungen. In der Mitte wird die KV Divergenz der beiden Normal Verteilungen dargestellt. Rechts ist die JS Divergenz der Beiden dargestellt. Man sieht sehr gut das asymmetrische Verhalten der KV und das symmetrische der JS. Dadurch lassen sich aussagekräftigere Gradienten, bestimmen welche zum Optimieren von D und G benötigt werden($?, ?$).

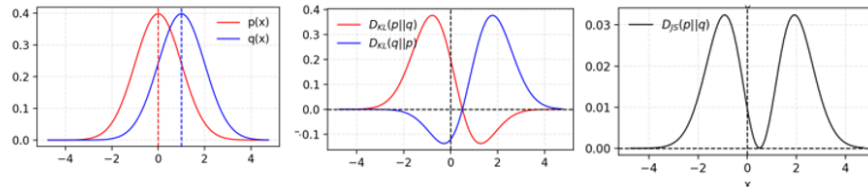


Abb. 25. KL Divergenz und JS Divergenz

2.6.1 Probleme mit Generative Adversarial Networks Wie auch die anderen generativen Modelle haben auch GANs noch Schwächen bezüglich der Trainingsabläufe und der Qualität der generierten Daten. Im Folgenden wird auf einige Probleme eingegangen.

Equilibrium D und G betreiben ein MinMax Spiel. Beide versuchen das Nash Equilibrium zu finden. Dies ist der bestmögliche Endpunkt in einem nicht kooperativen Spiel. Wie in dem Fall von GAN wäre das wenn $p_g = p_r$. Es wurde gezeigt, dass das Erreichen dieses Punktes sehr schwierig ist, da durch die Updates der Gewichte mit den Gradienten der Loss-Funktion starke Schwingungen der Funktion entstehen können. Dies kann zur Instabilität für das laufende Training führen (? , ?).

Vanishing gradient Dies beschreibt das Problem, wenn D perfekt trainiert ist mit $D(x) = 1, \forall x \in p_r$ und $D(x)=0 \forall x \in p_g$. Die Loss-Funktion würde in diesem Fall auf 0 fallen und es gäbe keinen Gradienten, für den die Gewichte von G angepasst werden können. Dies verlangsamt den Trainingsprozess bis hin zu einem kompletten Stopp des Trainings. Würde D zu schlecht trainiert mit $D(x) = 0, \forall x \in p_r$ und $D(x)=1 \forall x \in p_g$. Bekommt G kein Feedback über seine Leistung bei der Datengeneration hat er keine Möglichkeit p_r zu erlernen (? , ?).

Mode Collapse Während des Trainings von GAN kann es dazu kommen, dass der Generator möglicherweise auf eine Einstellung seiner Gewichte fixiert wird und es zu einem sogenannten Mode Collapse führt. Was zur Folge hat, dass der Generator sehr ähnliche Samples produziert (? , ?).

Keine aussagekräftigen Evaluations Metriken Die Loss Funktion der GANs liefert keine aussagekräftigen Evaluationsmöglichkeit über den Fortschritt des Trainings. Bei discriminativen Modellen im üblichen Machine Learning besteht die Möglichkeit Validierungsdatensätze zu verwenden und an diesen die Genauigkeit des Modells zu testen. Diese Möglichkeit besteht bei GANs nicht(? , ?).

2.6.2 Lösungsansätze für Generative Adversarial Networks Probleme

Nun werden einige Techniken aufgezeigt, welche die unter Abschnitt Probleme mit GAN genannten Schwierigkeiten angehen und zu einem effizienteren Training führen, damit eine schnellere Konvergenz während des Trainings erreicht wird.

Feature matching Dies soll die Instabilität von GANS verbessern und gegen das Problem des Vanishing Gradient angehen. G bekommt eine neue Loss-Funktion und ersetzt die des üblichen Vanilla GAN. Diese soll G davon abhalten, sich an D über zu trainieren und sich zu sehr darauf zu fokussieren, D zu täuschen und gleichzeitig auch versuchen die Datenverteilung der Trainingsdaten abzudecken(?, ?).

Minibatch discrimination Um das Problem des Mode Collapse zu umgehen, so dass es nicht zu einem Festfahren der Gewichte von G kommt, wird beim Trainieren die Nähe von den Trainingsdatenpunkten gemessen. Anschließend wird die Summe über der Differenz aller Trainingspunkte genommen und dem Discriminator als zusätzlicher Input beim Training hinzugegeben (?, ?).

Historical Averaging Beim Training werden die Gewichte von G und D aufgezeichnet und je Trainingsschritt i verglichen. Anschließend wird an die Lossfunktion je Trainingsschritt die Veränderung zu $i-1$ an die Loss-Funktion addiert. Damit wird eine zu starke Veränderung bei den jeweiligen Trainings-schritten bestraft und soll gegen ein Model Collapse helfen (?, ?).

One-sided Label Smoothing Die üblichen Label für den Trainingsdurchlauf von 1 und 0 werden durch die Werte 0.9 und 0.1 ersetzt. Dies führt zu besseren Trainingsergebnissen. Es gibt derzeit nur empirische Belege für den Erfolg, jedoch nicht weshalb diese Technik besser funktioniert(?, ?).

Adding Noises Noise an den Input von D zu hängen kann gegen das Problem des Vanishing gradienten helfen und das Training verbessern(?, ?).

Use Better Metric of Distribution Similarity Die JS Divergenz von vanilla GAN sorgt für bessere Trainingsergebnisse im Vergleich zu der KL Divergenz von anderen generativen Modelle. Jedoch weißt die JS immernoch Probleme auf. Es wird vorgeschlagen diese durch die Wasserstein Metric zu ersetzen, da diese bessere Ergebnisse bei disjunkten Wahrscheinlichkeitsverteilungen liefern kann(?, ?).

2.6.3 Conditional Adversarial Networks Conditional Adversarial Networks (CGAN) beruhen auf dem Grundkonzept von vanilla GAN (vgl. Kapitel GAN). Es wird zusätzlich die extra Information y hinzugefügt. Diese kann jegliche Information sein, welche auf x abgestimmt ist. Beispielsweise kann y ein Klassenlabel zu den gelernten $P(x)$ sein. Die Zielfunktion des CGAN ist

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

und beschreibt eine bedingte Wahrscheinlichkeit, dass ein Trainingsdatensatz x oder ein Datensatz, welcher von dem Generator erzeugt wurde, von y abhängt ($?, ?$).

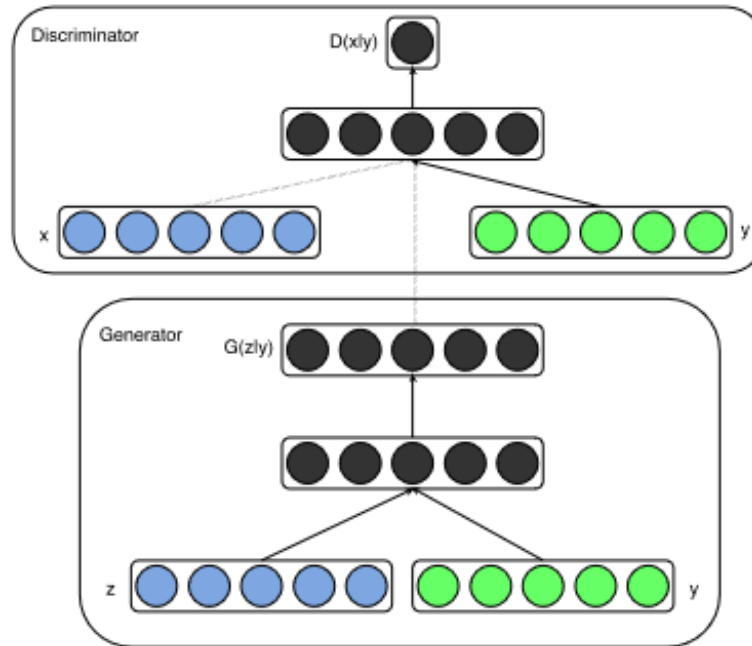


Abb. 26. Conditional Adversarial Network

Ein beispielhafter Anwendungsfall ist das Erlernen von selbstgenerierten Zahlen. Der MNIST Datensatz besteht aus 10000 handschriftlich eingescannten Zahlen von 0 - 9. Die Daten x können nun während des Trainings mit den dazugehörigen Zahlenlabels y trainiert werden und anschließend kann der Generator verwendet werden, um selbst die gewünschten x zu erzeugen, indem y gewählt wird ($?, ?$). Das Konzept von CGAN wird nun in folgender Arbeit weiterentwickelt, um das Ziel zu haben, selber Bilder zu verändern.

2.7 Conditional-GAN

Conditional-GAN(C-GAN) ist eine Modifikation des ursprünglichen GAN von Goodfellow, welches erlaubt bedingte Wahrscheinlichkeiten in Datensätze zu erlernen. Das heißt zusätzliche Informationen in den Lernprozess einzuspeisen um den Output zu modifizieren. Im ursprünglichen GAN gibt es keine Möglichkeit auf den Output des GANs Einfluss zu nehmen. Dabei wird das Modell so verändert das eine zusätzliche Information y zusätzlich als Input in den Discriminator und den Generator zugefügt wird. Dabei kann y jedliche Information sein wie Label, Bilddaten oder 3D-Daten. Dem entsprechend muss die Zielfunktion dahin gehen angepasst werden bedingte Wahrscheinlichkeiten zu lernen

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

Im Abb. 27 kann der Informationsfluss und die Konnektivität der einzelnen Module entnommen werden. Die Module Generator und Discriminator bleiben gleich und können von ihren Aufbau für die jeweiligen Datentyp verändert werden. Zum Trainingsablauf können die gleichen Abläufe wie in Kapitel GAN TRICKS verwendet werden um das Trainingsergebnis zu verbessern und die Trainingsdauer zu verkürzen.

2.8 3D-GAN

Das besondere an den 3D Raum im Vergleich zu Normalen 2D Bildern ist die hohe Steigerung der Dimension und zu gleich der Hohe Informationsgehalt welcher in 3D Objekten steckt. Das Ziel von 3D-GANS ist es Modelle von Objekten zu erhalten. Dabei wird der latente Objektraum erfasst und soll damit die Wahrscheinlichkeiten für einzelne Objektklassen enthalten.

Die Architektur des typischen 3D-GAN ist dem vanilla GAN von Goodfellow ähnelt. Durch den Aufbau von 3D-Objekten unterscheidet sich nur die Dimensionalität der einzelnen Layer. Dadurch das 3d Objekte ins Räumliche Verhältnis gestellt werden müssen braucht man eine dritte Dimension in den Layern welche die Tiefe der Objekte darstellt. (?, ?).

Die Zielfunktion bleibt die gleiche wie bei den üblichen GANS $\min_G \max_D V(D, G) = E_{x \sim p_{\text{data}}(x)} [\log D(x|y)] + E_{z \sim p_z(z)} [\log(1 - D(G(z|y)))]$ (?, ?)

Das latent GAN hingegen benutzt eine andere Technik. Zunächst wird ein Autoencoder(siehe Autoencoder) verwebdet um Trainingsdaten zu trainieren. Ziel dabei ist den latenten Raum der Trainingsdaten zu erlernen und eine Kompression der Daten um den Suchraum zu verringern und dadurch das Training des GANs zu erleichtern. Nachdem das Trainingsabgeschlossen hat wird der Trainingsdatensatz durch den Encoder komprimiert und das Training vom GAN wird mit dem komprimierten Datensatz getan.

(?, ?)

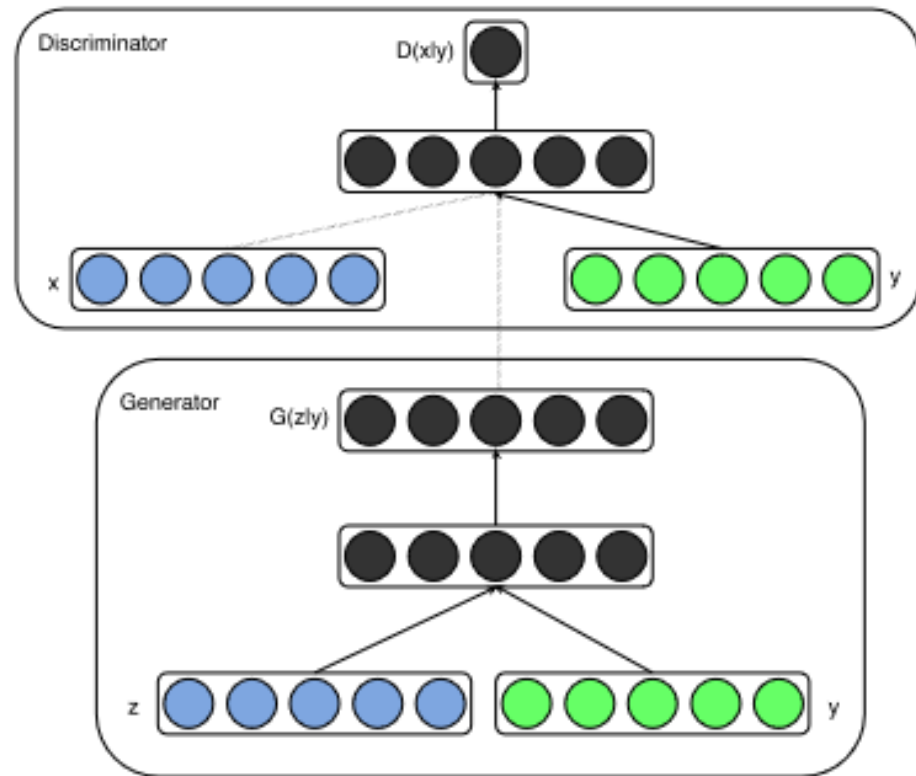


Abb. 27. Conditional Adversarial Network

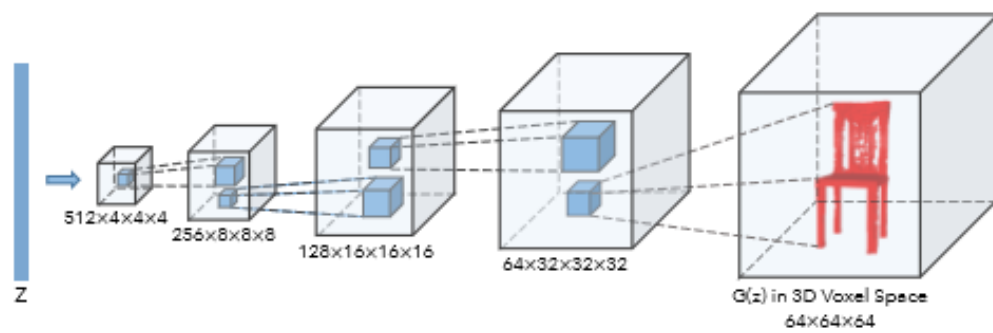


Abb. 28. Conditional Adversarial Network

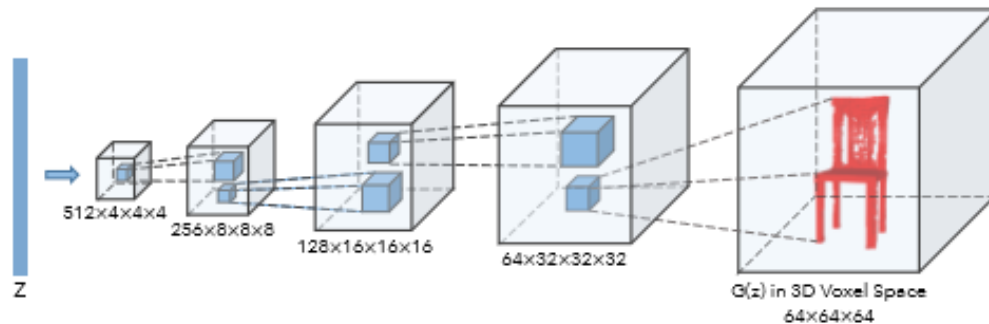


Abb. 29. Conditional Adversarial Network

3 Methoden

In diesem Kapitel werden die Methoden für die Versuchsaufbauten besprochen, welche die Ziele 1 - 3 überprüfen sollen. Es wird zunächst ein genereller Struktur des Aufbaus besprochen und anschließend auf spezifische Gegebenheiten eingegangen. Außerdem werden die spezifischen Datensätze für die jeweiligen Versuchsaufbauten aufgezeigt und ihre Entstehung erläutert.

Alle Versuche wurden auf einen Computer mit Ubuntu 14.05 Betriebssystem mit einem Intel® Core™ i7-7700k mit 4.50GHz und einer GeForce GTX 1089 mit 8GB Grafikkarte. Training und Testen wurden mit CUDA 9.0 und cudNN 7.1.1.

3.1 Aufbau

3.1.1 Datensatz 1. Versuchsaufbau Der erste Datensatz "Stühle" besteht aus 4014 Punktwolken mit je 2056 Punkten. Der Datensatz wurde aus dem ShapeNet Datensatz entnommen. Ein Beispieldatensatz kann aus Abbildung entnommen werden. Er ist als .ply Datenformat abgespeichert.

Die Daten für den zweiten Datensatz "Blätterstämme" vom Fraunhofer Institut. Der Grunddatensatz bestand aus mehreren 3D-Scans von Tabakpflanzen, bei denen die Blätter der Pflanze zu einem Datensatz zusammengefügt wurden sind. Der "Blätter"-Datensatz besteht aus 420 Punktwolken mit je 2056 Punkten. Er ist als .ply Datenformat abgespeichert.

3.1.2 Datensatz 2. Versuchsaufbau Für den zweiten Versuchsaufbau wird auf den in Datensatz Versuchsaufbau 1. Blatt Datensatz "Blätterbüschel" zugegriffen. Dabei werden in den Blättern Punkte herausgenommen, welche das Verdecken oder Zerstören eines Blattes simulieren sollen. Dies geschieht, indem eine Gaußverteilung auf einer 3D-Sphäre erzeugt wird. Anschließend wird das Komplement zwischen einem 3D-Blatt und einer 3D-Sphäre berechnet. Das Ergebnis sind zerstörte Blätter mit kreisförmigen Löchern auf der Oberfläche. Der Algorithmus zur Erzeugung der Trainingsdaten kann im Anhang entnommen werden.



Abb. 30. 3D Punktwolke einer Tabakpflanze

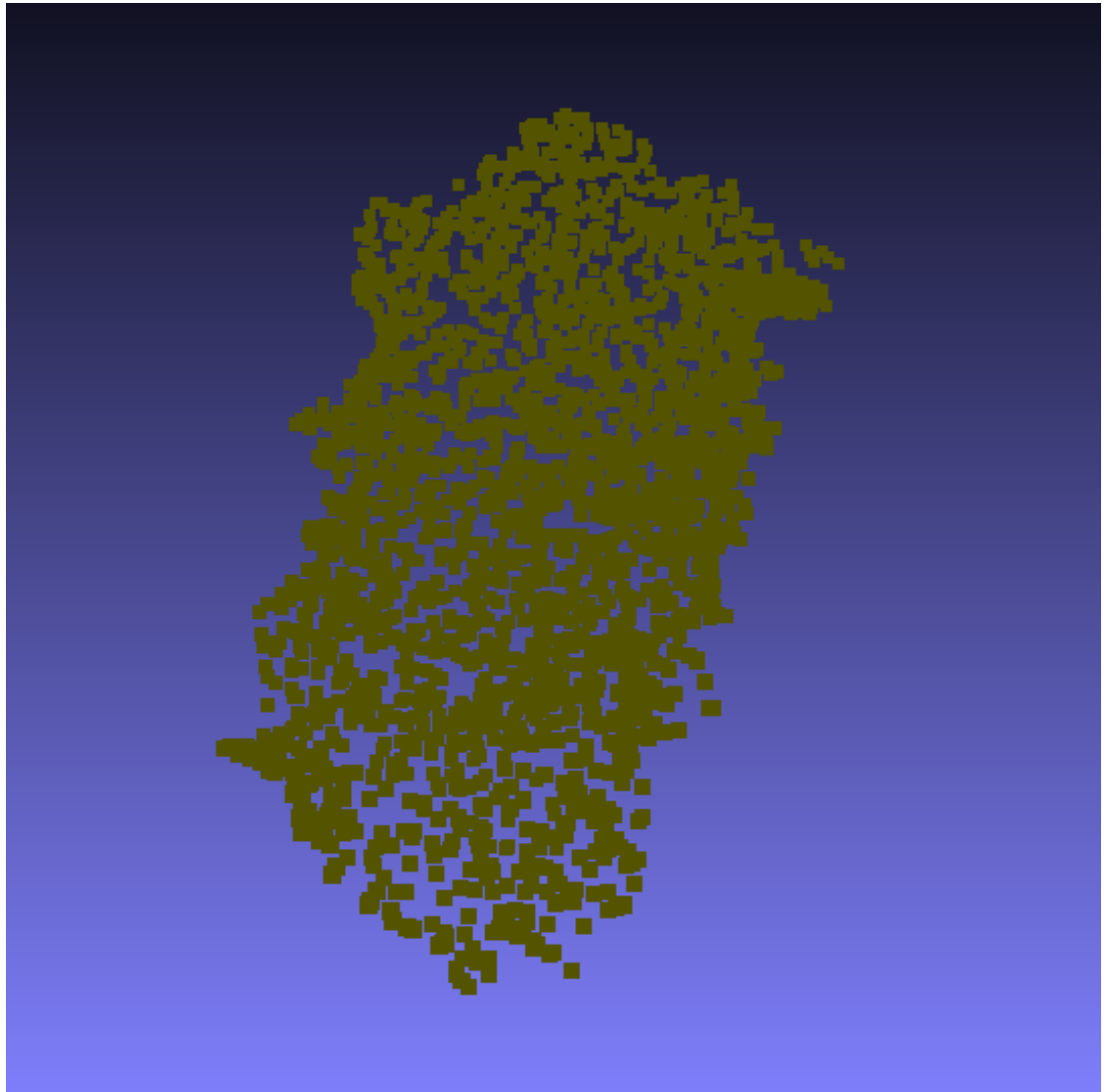


Abb. 31. 3D Punktwolke einer Tabakpflanze

In Abbildung sind die 3 Abschnitte zu erkennen wie die Trainingsdaten erzeugt werden.

Insgesamt besteht der Trainingsdatensatz aus 450 unzerstörten Blättern sowie 1024 zerstörten paaren.

3.1.3 Trainingsaufbau 1 - GAN Der Aufbau besteht aus zwei unterschiedlichen Versuchen. Aufbau 1 ist der RAW - GAN Aufbau. Dabei wird auf den herkömmlichen Aufbau von GAN zurückgegriffen. Die Allgemeinen Meta-Trainingsvariablen sind Learningrate mit 0,005 und einen AdamOptimizer, mit einem Beta1 von 0.5 und einem Beta2 von 0,5. Der Discriminator besteht aus 4-Layern welche 1-Dimensionaler Convolutional Layer bestehen. Mit einer Kernel Größe von 1 und stride von 1. Die Aktivierungsfunktion sind Relu. Darauf folgt 3 fully connected Layer mit der Größe 128, 54 und 1 alle mit einer Relu Aktivierungsfunktion. Der Generator besteht auf 5 fully-connected Layern mit [64,128,512,1024,1536] neuronen jeweils mit einer Relu Aktivationsfunktion. Der Aufbau kann in Abbildung entnommen werden. Die Zielfunktion ist die Wasserstein Metric und die übliche GAN Zielfunktion.

Der Versuchsaufbau 2 ist das Latent-GAN zunächst wird dabei der Autoencoder mit den Trainingsdaten trainiert. Der Encoder des Autoencoders wird mit einer Learning Rate von 0.0005 trainiert. und einer Batch Größe von 50. Der Encoder besteht dabei besteht dabei aus 4 1-D Convolutional Layern mit [64,128,256,1024] Filtern Stride von 1 und Size von 1. Die Aktivierungsfunktion ist relu. Der Letzte Layer ist ein Max Layer. Der Decoder besteht aus aus [256,256,[614]

3.1.4 Trainingsaufbau 2 - CGAN für Punktwolkenrekonstruktion

Da sich im Trainingsaufbau 1 das L-GAN bessere Ergebnisse liefert bei Erlernen von Punktwolken Daten. Wird der Aufbau von L-GAN übernommen und dahin gehen verändert, das Ziel zu erfüllen. Zunächst werden wie bei Trainingsaufbau 1 die Trainingsdaten (siehe Trainingsdaten Aufbau 2) trainiert. Der Aufbau des Autoencoders ist dabei gleich dem von Aufbau 1. Der Encoder des Autoencoders wird mit einer Learning Rate von 0.0005 trainiert. und einer Batch Größe von 50. Der Encoder besteht dabei besteht dabei aus 4 1-D Convolutional Layern mit [64,128,256,1024] Filtern Stride von 1 und Size von 1. Die Aktivierungsfunktion ist relu. Der Letzte Layer ist ein Max Layer. Der Decoder besteht aus aus [256,256,[614]. Die Zielfunktion ist die Chamfer Distance da sie die besten Ergebnisse bei citiere PGAN besten geliefert hat.

Anschließend werden zerstörten Blatt Daten x mit den trainierten Encoder zu den latenten Code y komprimiert, der einen Vektor von 128-D entspricht. Das selbe wird mit den unzerstörten Blatt Daten x' gemacht welche mit den Encoder zu dem latenten Code y' komprimiert werden welcher einen Vektor von 128-D entspricht. Wobei es jeweils (x, x') paare Gibt welche den

Der Aufbau des C-GANs entspricht der in im 3.4 beschriebenen C-GAN üblichen Aufbau. der Generator bekommt als Input x' und x welches einen 128-D Vektor

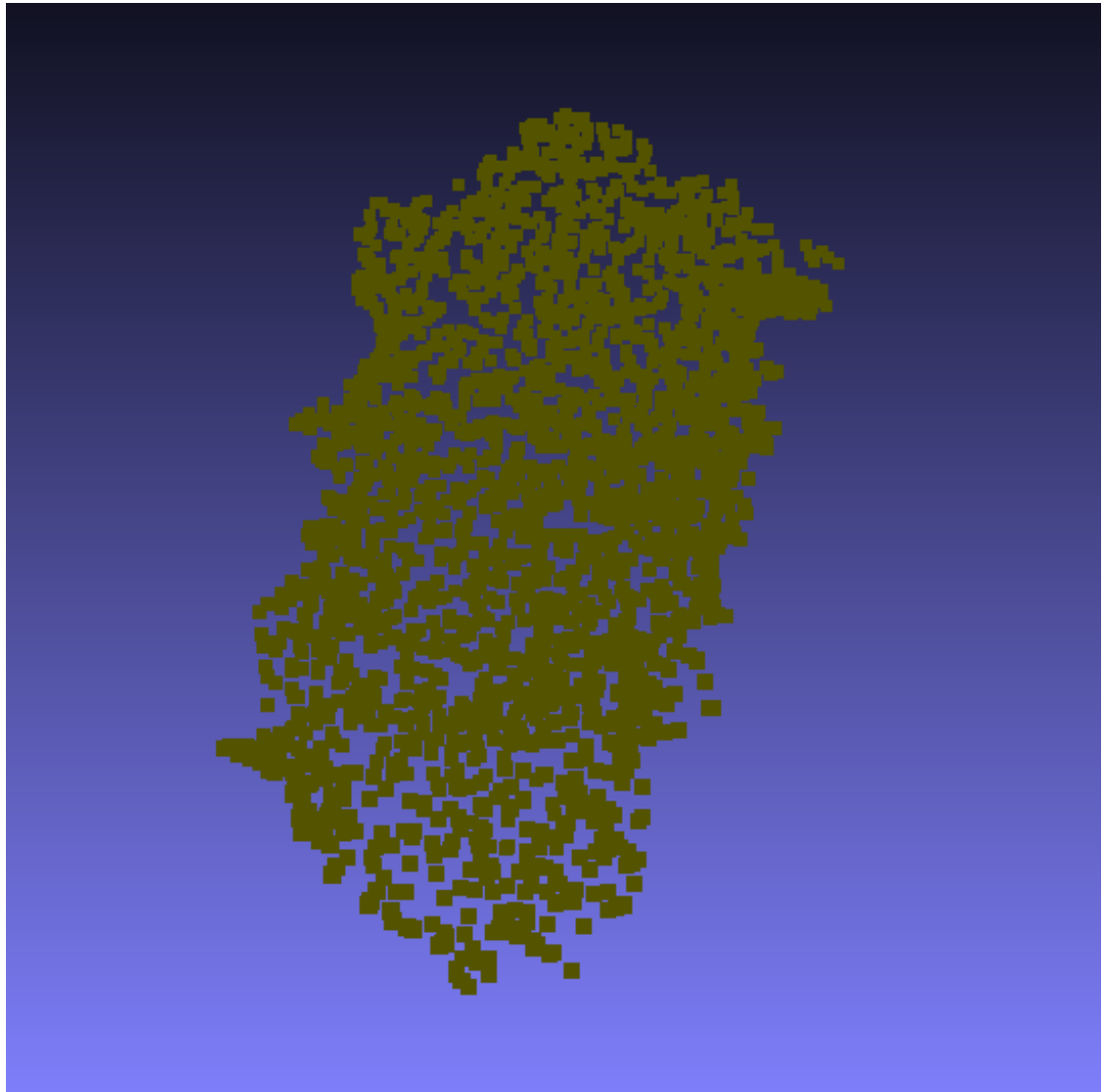


Abb. 32. 3D Punktwolke einer Tabakpflanze

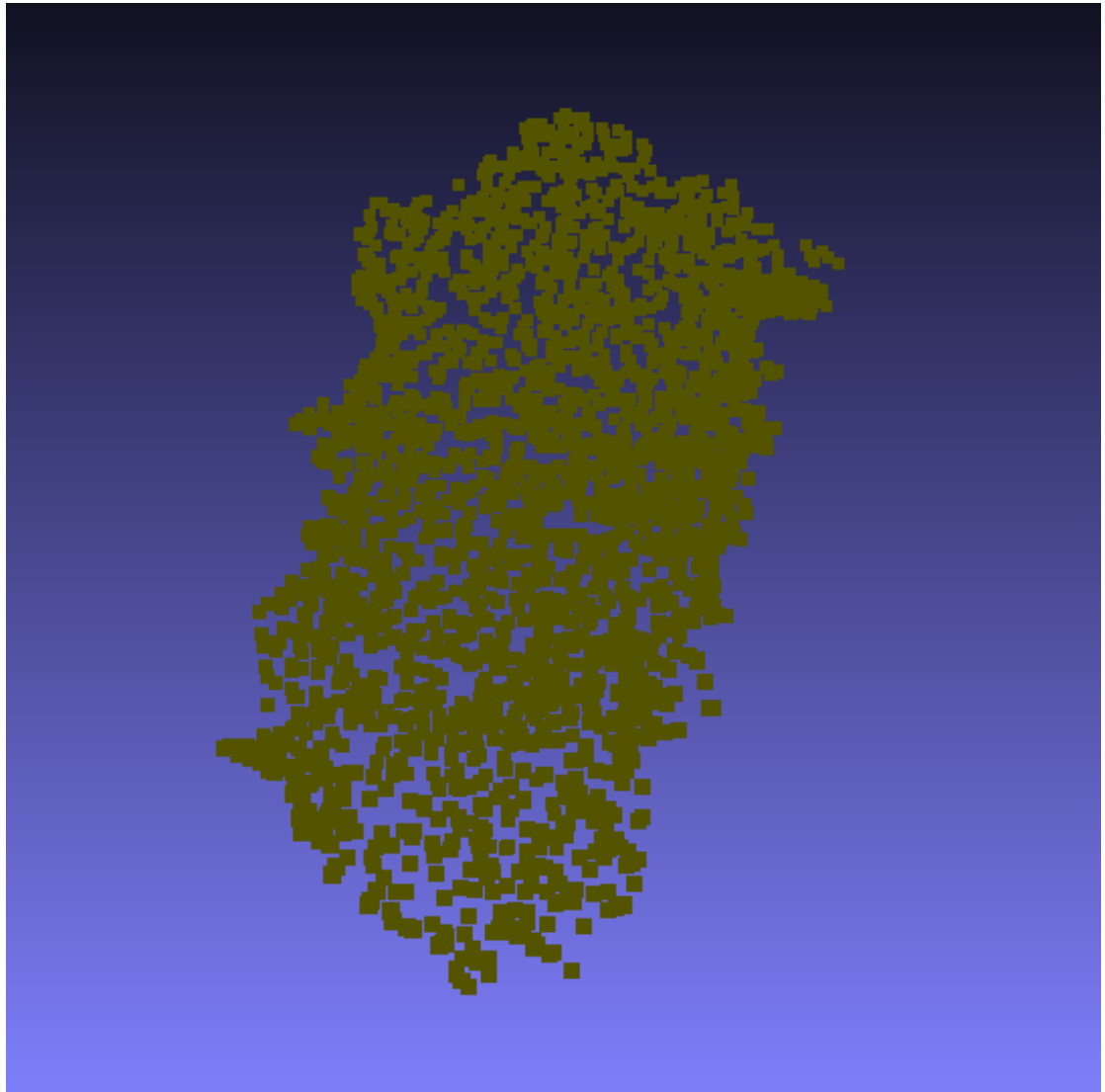


Abb. 33. 3D Punktwolke einer Tabakpflanze

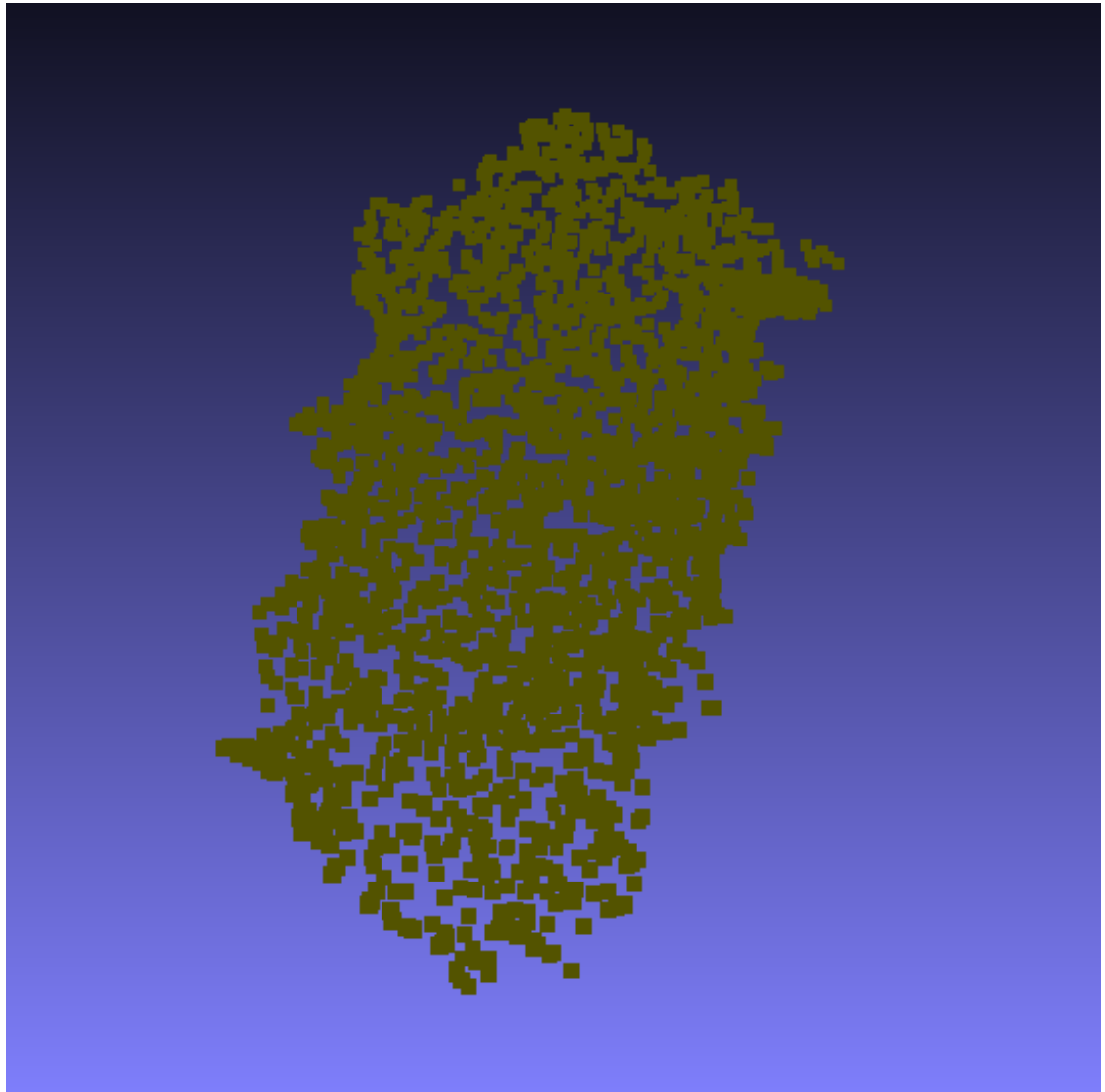


Abb. 34. 3D Punktwolke einer Tabakpflanze

welcher aus einer Gausischen Verteilung gesamplet wird. Der Generator besteht aus 3 Layern [256,128,128] der Diskriminator besteht aus 2 fully Connected Layer mit der Größe [128,128]

Bei Generator und Diskriminator wird beim Trainieren jeweils Batchnormalization eingesetzt.

Als Zielfunktion wird wieder das W-GAN verwendet da es im Versuchsaufbau 1 die besseren Ergebnisse liefert.

3.2 Ergebnis

Es wird nun in diesem Kapitel auf die Ergebnisse von Versuchsaufbau 1 und 2 eingegangen. Zunächst wird Versuchsaufbau 1 mit dem Erlernen von latenten Raum von Punktwolken gezeigt. Anschließend geht es um die Ergebnisse von Versuchsaufbau 2 für die Rekonstruktion von zerstörten Blättern.

3.2.1 Ergebnisse - Versuchsaufbau 1

3.2.2 Ergebnisse - Versuchsaufbau 2

4 Evaluation und Ergebnisse

Bilder aus nicht Trainingsdatensatz

5 Zusammenfassung und Diskussion

Ein GAN besteht aus zwei ANN, dem Discriminator und dem Generator. Das Ziel des G ist es Daten zu erzeugen, welche nicht von Trainingsdaten unterschieden werden können. Der Discriminator hat die Aufgabe zu unterscheiden ob der jeweilige Datensatz von G erzeugt wurde und somit ein fake Datensatz ist, oder von den Trainingsdaten stammt(?). GANs werden derzeit noch erforscht. Es gibt noch einige offene Fragen, beispielsweise bezüglich der Performance hochauflösender Bilder(?). Es wurden in diesem Paper einige Probleme, welche beim Trainieren von GAN auftreten können und mögliche Lösungsansätze, vorgestellt. Es gibt derzeit einige praktische Ansätze, welche in der Anwendung auf GANs zurück greifen. Beispielsweise durch Textbeschreibung eigene Bilder als Output generiert werden (?), oder 3D Daten erzeugt werden(?). GANs finden Anwendung in unterschiedlichen Bereichen des Deep Learnings, da sie als Lösung des Problems angesehen werden, dass Neuronale Netzwerke eine hohe Menge an Trainingsdaten benötigen und GANs dieses Problem durch ihre Fähigkeit, neue Daten zu generieren, umgehen. GANs lernen eine Art "versteckte Repräsentation von Klassen, was dazu beitragen kann auch Modelle von komplexen Prozessen zu erlernen. Es gibt erste Ansätze bei denen im Reinforcement Learning durch GANs versucht wird Modelle von der Umwelt eines Agenten zu erlernen, welche dann auf Grundlage dieser Modelle Vorhersagen über zukünftige Ereignisse treffen kann(?).

Abbildungsverzeichnis

1	künstliches Neuron	6
2	künstliches neuronales Netzwerk	7
3	Sigmoid Funktion	8
4	Softmax	8
5	Minimum Maximum Saddle Point	9
6	LossFunktion	10
7	aaaa	13
8	Nicht trennbar	15
9	Sigmoid Funktion	16
10	Softmax	16
11	Punktwolke eines Tabakblattes	18
12	Polygon File Format	19
13	TERRA-REF Feld Scanner	19
14	Scankopf	20
15	Neuronal Network als Markov Chain	21
16	Convolutional Neural Network	22
17	Convolution Beispiel	22
18	Deconvolution Beispiel	23
19	autoencoder	24
20	Generative Modelle	26
21	Generative Modelle	27
22	Generative Modelle	27
23	Generativ Adversarial Network	28
24	Deep Convolutional GAN	30
25	KL Divergenz und JS Divergenz	31
26	Conditional Adversarial Network	33
27	Conditional Adversarial Network	35
28	Conditional Adversarial Network	35
29	Conditional Adversarial Network	36
30	3D Punktwolke einer Tabakpflanze	37
31	3D Punktwolke einer Tabakpflanze	38
32	3D Punktwolke einer Tabakpflanze	40
33	3D Punktwolke einer Tabakpflanze	41
34	3D Punktwolke einer Tabakpflanze	42

Tabellenverzeichnis

Literaturverzeichnis