

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	8
2.1.2 Momentum	11
2.1.3 Batch Normalisation	11
2.1.4 Ziel-Funktion	12
2.2 Datenformat	14
2.2.1 Woher stammen die Daten	14
2.2.2 Das Problem mit 3D-Data bei Machine Learning Ansätzen	17
2.3 Convolution Neural Networks	17
2.4 Autoencoder	20
2.5 Generative Adversarial Network	22
2.5.1 Probleme mit Generative Adversarial Networks	26
2.5.2 Lösungsansätze für Generative Adversarial Networks	
Probleme	28
2.6 Conditional-GAN	29
2.7 3D-GAN	29
2.8 Rekonstruktion von Daten	32
3 Methoden	32
3.1 Aufbau	33
3.1.1 Datensatz 1. Versuchsaufbau	33
3.1.2 Datensatz 2. Versuchsaufbau	33
3.1.3 Trainingsaufbau 1 - GAN	33
3.1.4 Trainingsaufbau 2 - CGAN für	
Punktwolkenrekonstruktion	37
4 Evaluation und Ergebnisse	37
4.1 Ergebnisse - Versuchsaufbau 1	37
4.2 Ergebnisse - Versuchsaufbau 2	38
5 Zusammenfassung und Diskussion	45
Abbildungsverzeichnis	50

Tabellenverzeichnis	52
----------------------------------	----

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 vorrangegeganngen 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 werden auf theoretische Grundlagen eingegangen, welche zum Verständnis dieser Arbeit benötigt wird. Zunächst werden generative Modelle im Allgemeinen vorgestellt, welche den Grundgedanken der Datengeneration für GANs aufzeigen und mit deren Hilfe es möglich gemacht werden soll

unkenntliche beziehungsweise beschädigte Modelle von Objekten wieder herzustellen. Diese Theorie baut zunächst auf die Datenstruktur von künstlichen Neuronalen Netzwerken auf, welche durch verschiedene Modelle dargestellt werden können. Im Kapitel Conditional-GAN und 3D-GAN werden die theoretischen Grundbausteine der vorherigen Themen vertieft und bilden den Grundstein für das in dieser Arbeit vorgestellte verfahren zur 3D-Datenrekonstruktion von 3D-Modellen aus Tabakblättern.

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. Diese Approximation wird durch Machine Learning Ansätzen erlernt man spricht auch von einem Optimierungsproblem. KNN können aus mehreren Schichten sogenannten Hidden Layer n besteht welche als $f^{(n)}$ dargestellt werden und $n \geq 1$ sein muss. Ein 2-Layer KNN ist dann definiert durch $f(x) = f^{(2)}(f^{(1)}(x))$. Es gibt einen Input Layer welche den Input in das Netzwerk aufnimmt. Ein ANN mit mehr als nur 1 Layer und wird dies dies mit Maschine Learning trainiert 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 ist das $f^{(2)}(?, ?)$. In 7 sind ein ANN exemplarisch dargestellt. Es soll die Konnektivität der einzelnen Layer veranschaulichen und den Informationsfluss von Input zu Output.

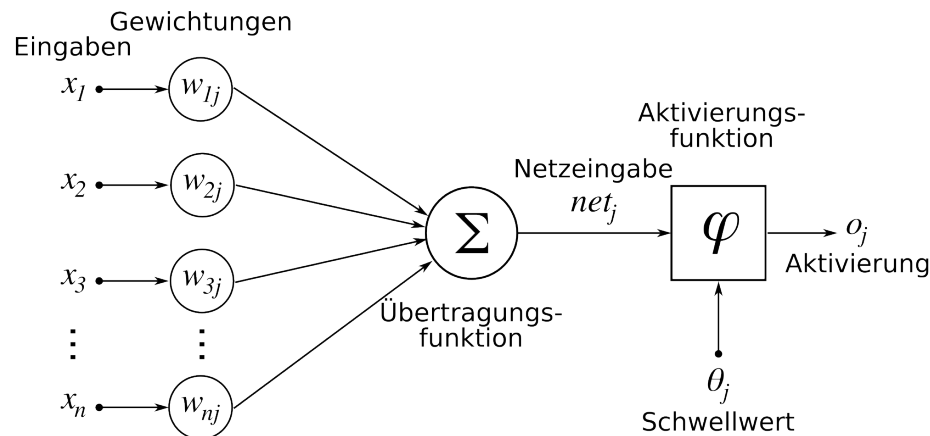


Abb. 1. künstliches Neuron

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.

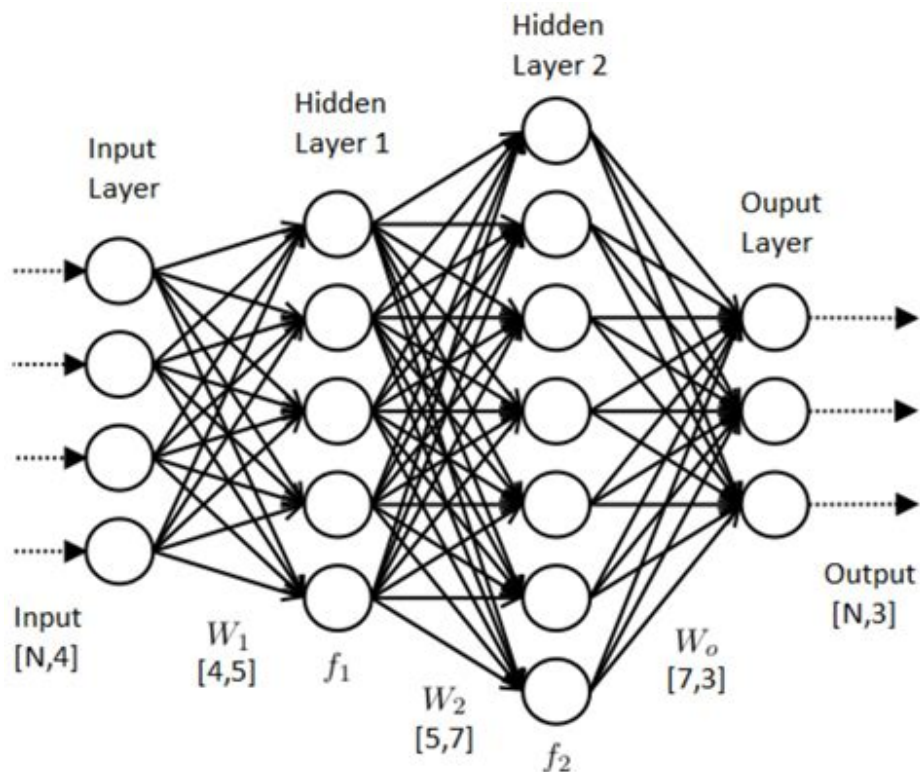


Abb. 2. künstliches neuronales Netzwerk

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. Ein Neuron ist definiert durch:

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

Die Aufgabe der Aktivierungsfunktion ist es eine nicht lineare Transformation des Inputs zu erzeugen. Damit kann das KNN nicht lineare Funktionen abbilden und somit komplexere Aufgaben lösen. Es gibt unterschiedliche Aktivierungsfunktionen Beispiele sind Sigmoid-Funktion $\sigma(x) = \frac{1}{1+\exp(-x)}$ oder Softmax-Funktion $\zeta(x)_j = \frac{e^{x_j}}{\sum_{k=1}^K e^{x_k}}$ oder Rectified Linear Unit (ReLU) $f(x) = \max(0, x)$

oder Leaky Rectified Linear Unit (Leaky Relu) $f(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{sonst} \end{cases}$ Der

Funktionsplot von Softmax und Sigmoid kann in Abb. 13 und 14 entnommen werden.

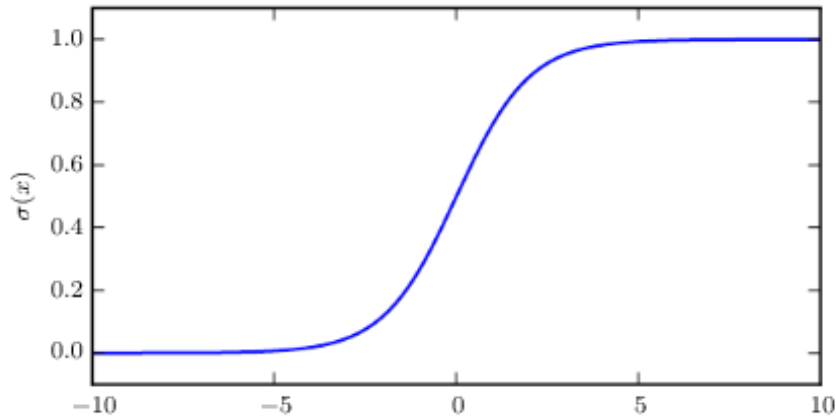


Abb. 3. Sigmoid Funktion

2.1.1 Backpropagation Algorithmus Um nun KNNs zu trainieren und den gewünschten Output y zu generieren wird der Backpropagation Algorithmus benutzt werden dieser zählt zu den Optimierungs- Algorithmen für KNN. Er konnte zeigen das dieser Algorithmus schneller und effizienter auf Neuronalen Netzwerken arbeitet als andere Optimierungsalgorithmen vor ihm. Das mathematische Zugrundeliegende Konzept ist ein Optimierungsproblem der die

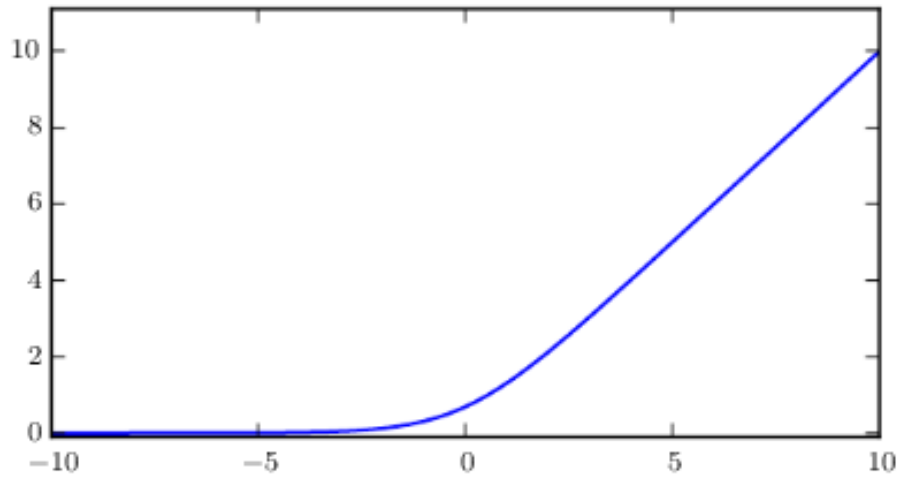


Abb. 4. Softmax

Partiellen Ableitung von $\frac{\partial Z}{\partial w}$, wobei Z die Zielfunktion und w die Gewichte im zu optimierenden Neuronalen 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. Durch die Steigung an Punkt x ist man nun in der Lage einer Änderung von der Ableitung x dahin gehen optimieren ($?$, $?$). Dieses Verfahren kann dabei helfen Funktionen das KNN dahingehen zu optimieren den gewünschten Output y zu erlangen.

Wenn nun $f'(x) = 0$ haben wir keinerlei Information über die Steigung erreicht. Dies ist aber kein Indiz dafür sein das f ein Optimum erreicht hat. Es könnten wie in Abbildung 5 unten dargestellt ein lokales Minimum sein das heißt das wir nur an diesen Punkt ein Minimum erreicht haben 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 optimieren in dem man in die Richtung des Gradienten geht welche negativ ist. Dieses Verfahren wird Gradient Descent genannt ($?$, $?$).

Algorithmen die für Deep Learning eingesetzt werden, beinhalten eine Art von Optimierung, ohne diese ist eine Umsetzung des Lernprozesses nahezu unmöglich. Diese Minimierungsfunktionen 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

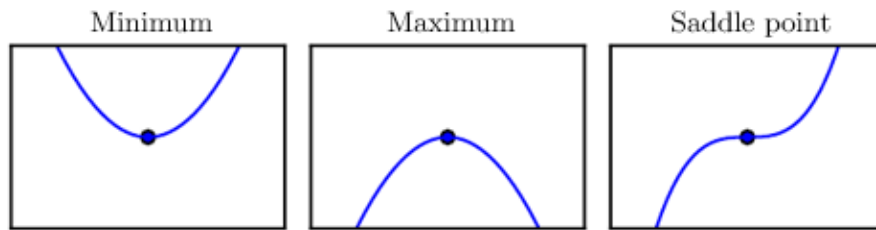


Abb. 5. Minimum Maximum Saddle Point

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 Gradient 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).

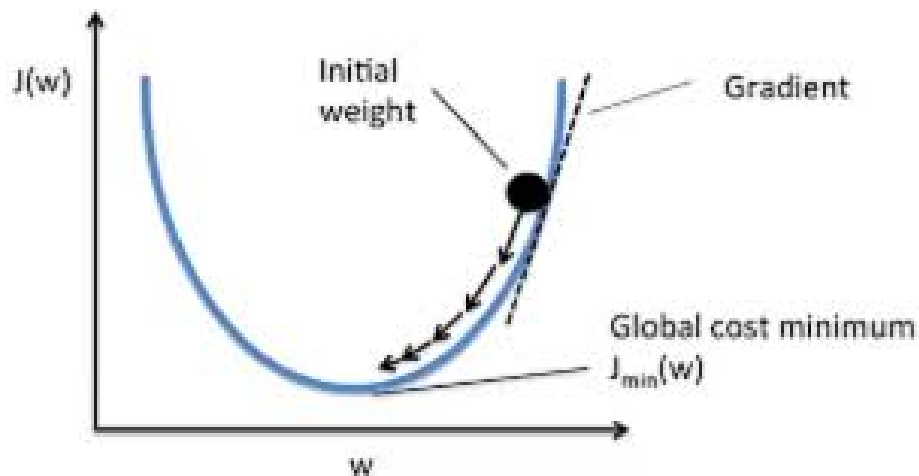


Abb. 6. LossFunktion

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 des Backpropagation Algorithmus zu beschleunigen um eine effizienter Ergebnis herbei zu führen und ein schnelleres Lernen erzielt. Definiert ist dieses durch:

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

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 einer Lernphase der Gradient üblicherweise hoch ist empfiehlt sich zunächst mit einem niedrigen Momentum zu arbeiten da sonst die Gefahr besteht über das globale Optimum hinweg zu schießen. Wenn nun das Training stagniert welches aus Gründen 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 je länger das Training voran schreitet. Und sich somit schneller zu einem globalen Optimum zu bewegen oder aber aus einem lokalen Optimum hinaus Richtung eines globalen Optimum(?). .

2.1.3 Batch Normalisation

Wie in Kapitel 2.1.1 gezeigt das stochastischen Gradienten Verfahren Vorteile gegenüber des normalen Gradienten Ermittlung beim Training von KNN. Dadurch das der Input in jeden Layer abhängig von den vorherigen Layern ist können Änderungen von Werten in frühen Layern des KNN große Auswirkungen in tieferen Layern im Netzwerk haben. Dadurch resultiert das in Trainingsabläufen die Verteilung der Gewichte in den jeweiligen Layern verlangsamt wird. Batchnormalization soll die Werteänderung von Gewichten verringern. Dieses Problem wird auch Covariance Shift genannt. Um dies zu verhindern zeigten Sergey Ioffe und Christian Szegedy (?, ?) eine Methode welche Batch Normalisation genannt wird. Je mehr Layer das Netzwerk hat desto stärker ist der Covariance Shift. Batch Normalisation besteht aus zwei Algorithmen. Algorithmus 1 verändert den eigentlichen Input von Layer n zu einem normalisierten Input y und Algorithmus 2 verändert das eigentliche Training eines 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 und In Schritt 3 die Varianze. In Schritt 3 wird nun der der normalisierte x_i berechnet welche dann mit β und γ multipliziert werden. Diese Werte sind neue Gewichte im Neurnonalen Netzwerk welche während des Trainingsprozesses 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 $\theta; \{x^{(k)}\}_{k=1} \text{ bis } N$
 - 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.

2.1.4 Ziel-Funktion Die Ziel-Funktion oder auch Loss-Funktion genannt $J(\theta)$ muss Differenzierbar sein. Das heißt unseren Funktion $f: D \rightarrow R$ ist differenzierbar an der Stelle x_0 . 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 Model θ $f^*(x)$ approximiert. Man verwendet auch gerne den Begriff Kosten, also wie viel Kosten unser Modell erzeugt bei den lösen der zugewiesenen

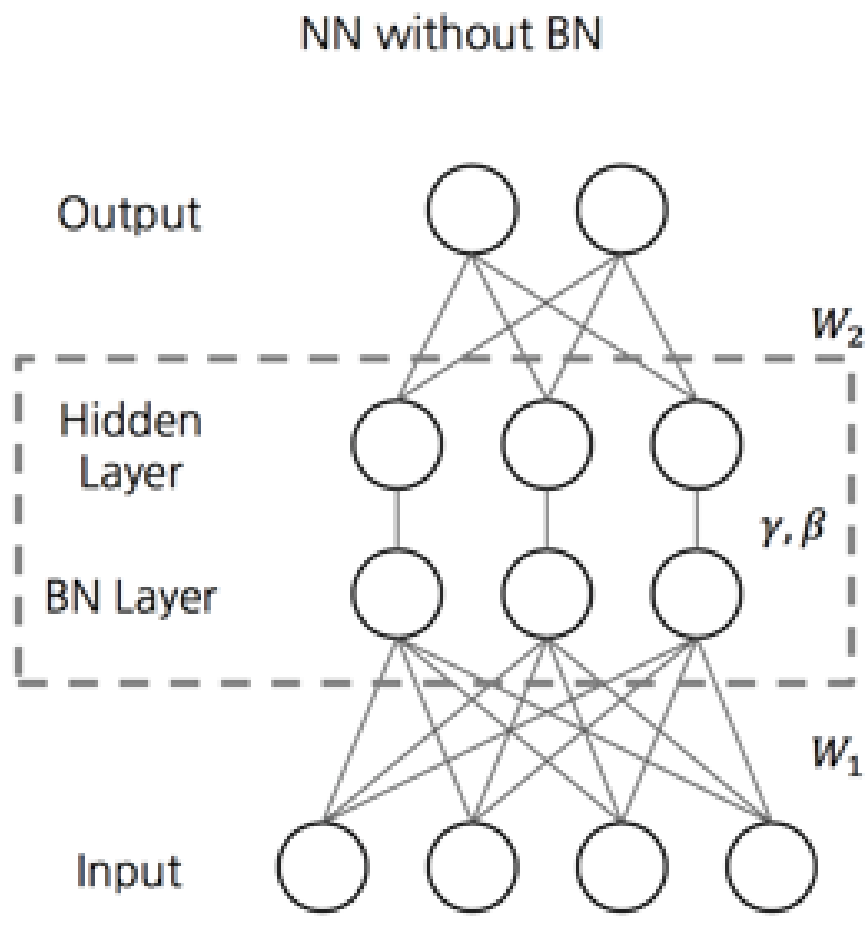


Abb. 7. Neuronales Netzwerk mit Batch Normalization Layer

Aufgabe. Die Wahl welche Funktion gewählt wird ergibt sich aus der Aufgabe des KNN. Diese kann beispielsweise für Klassifikation oder 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 Cross Entropy Loss Function vorgestellt (?, ?). Diese kann verwendet werden um Beispielsweiße Klassifikationsprobleme zu lösen. Diese ist definiert als:

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

Wobei $x_n:n=1,...,N$ die Trainingsdaten sind ist und $c_n:n=1,...,N$ die mögliche Klassen. Der Output ist die Wahrscheinlichkeit zwischen 0 und 1 ob $x_i \in$ der bestimmten Klasse enthalten ist. Auch kann sie hergenommen um zu messen wie hoch die Differenz zwischen zwei Wahrscheinlichkeitsverteilungen ist.

2.2 Datenformat

Punktwolken sind eine Menge von N Punkten welche im Vektorraum dargestellt werden können. Jeder Punkt $n \in N$ wird durch seine (x,y,z) Koordinaten 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. 10 kann eine Beispieldatei entnommen werden in der dieser Aufbau dargestellt ist. Diese kann als Menge betrachtet werden die jeweiligen Punkte sind geordnet in dieser Listen gespeichert jedoch spielt es keine Rolle für den Punktwolken-compiler 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. 8 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. 10 ist dieser abgebildet. In Abb. 11 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 reflektion 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.

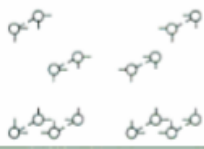
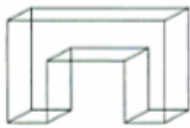
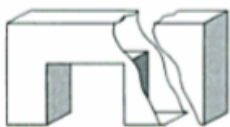

Dimension der Elemente	Element	Modelltyp
0D	<p>Punkt</p> 	Eckenmodell
1D	<p>Linie</p> 	Kantenmodell
2D	<p>Fläche</p> 	Flächenmodell
3D	<p>Volumen</p> 	Volumenmodell (Körpermodell)

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

Abb. 8. Punktwolke eines Tabakblattes


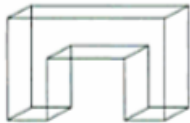
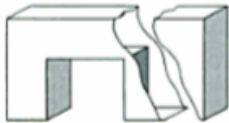

Dimension der Elemente	Element	Modelltyp
0D	<p>Punkt</p> 	Eckenmodell
1D	<p>Linie</p> 	Kantenmodell
2D	<p>Fläche</p> 	Flächenmodell
3D	<p>Volumen</p> 	Volumenmodell (Körpermodell)

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

Abb. 9. Polygon File Format



Abb. 10. TERRA-REF Feld Scanner

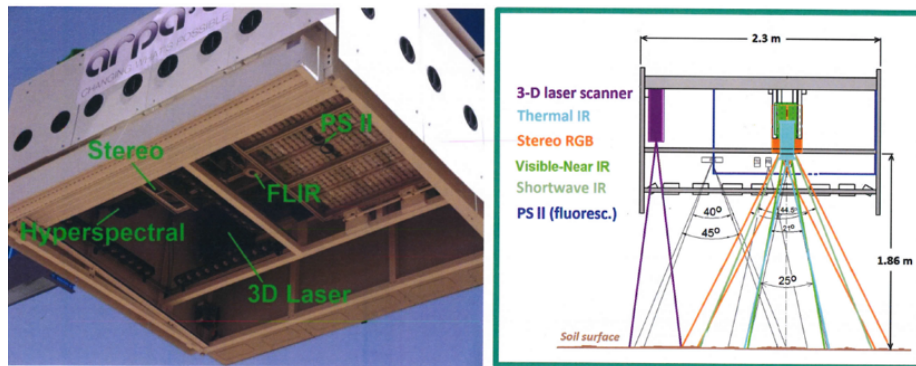


Abb. 11. Scankopf

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 steigt der Informationsgehalt und dadurch die Komplexität für das Anwenden von Maschine Learning Algorithmen enorm. Besonders bei Nicht-Euklidischen 3D-Daten wie Punktwolken deren keine Struktur zu Grunde liegt ist dieses gegeben (? , ?).

Wie im Kapitel "Woher stammen die daten "zeigt, sind Punktwolken als Menge gespeichert in denen keine Relation untereinander besteht, dass heißt es ist für den Punktwolkencompiler nicht von Relevanz auf welchen Platz die einzelnen Punkte abgespeichert werden, die Punktwolke wird egal in welcher Permutation der einzelnen Punkte abgespeichert, immer gleich angezeigt. Vergleicht man nun ein Bild mit 512 Pixeln und 3 RGB-Farbkanälen ist einer Dimension von 391680 erreicht. Vergleichen wir nun das mit einer Punktwolke in einen 125 cm³ großen Bereich. Da die einzelnen Koordianten eines Punktes als Rationale Zahlen dargestellt wird und rationalen Zahlen abzählbar Unendlich ist unser Suchraum üendlichzeichen "groß. Dies führt zu einen erheblichen mehr Aufwand für Machine Learning Ansätzen wie Deep Learning für Punktwolken.

Ein weitere Schwierigkeit die für Deep Learning mit Punktwolken ist, dass die aus Kapitel Convolutional Neural Network beschrieben Convolutionel-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 Punktwolken einzusetzen(? , ?).

2.3 Convolution Neural Networks

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

beispielsweise Bilder sein welche durch die Matrix $A = \begin{bmatrix} a_1 & a_2 & \dots & a_n \\ b_1 & b_2 & \dots & b_n \\ \vdots & n_n & \ddots & \vdots \\ x_1 & x_2 & x_3 & x_n \end{bmatrix}$

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 26 entnommen werden(?, ?).

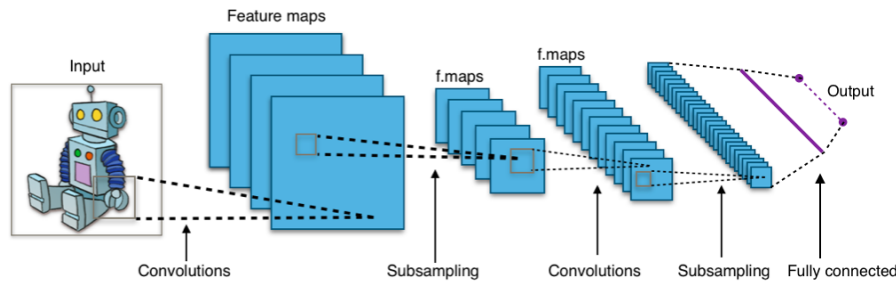


Abb. 12. Convolutional Neural Network

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 26 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 auf den Input und führt erneut einen Berechnungsschritt durch. Dieser Vorgang wird Stride genannt. An jeder Position wird das Produkt von jeden x_{ij} des Input und k_{ij} des Kernel durchgeführt. Anschließend werden alle Produkte aufsummiert. In Abbildung 13 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 jeweili-

gen 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 (?, ?).

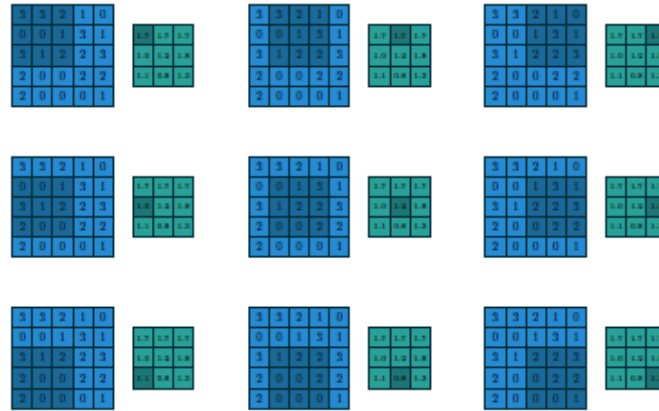


Abb. 13. Convolution Beispiel

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 7x7 und Stride 1. Der Input A für einen Kernel K ist ein 28x28 Matrix. Der Output aus diesen Filter sind 20 22x22 Feature Maps. Würde der Input ein 28x28x3 Bild mit 3 RGB Channels sein, der Output 60 22x22 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 x N definiert der Input I mit der Größe N x N und Stride $s = 1$. Deconvolution kann wie Convolution angesehen werden mit Zero Padding auf dem Input. Das in Abbildung 14 gezeigte Beispiel zeigt einen deconvolution Vorgang mit eine 3x3 Kernel über einen 4x4 Input. Dies ist gleich mit einen Convolution Schritt mit einen 3x3 kernel auf einen 2x2 Input und einer 2x2 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 NxN auf eine Output Größe KxK, wobei $K > N$ mit Abhängigkeit von Kernel und Stride abgebildet werden(?, ?).

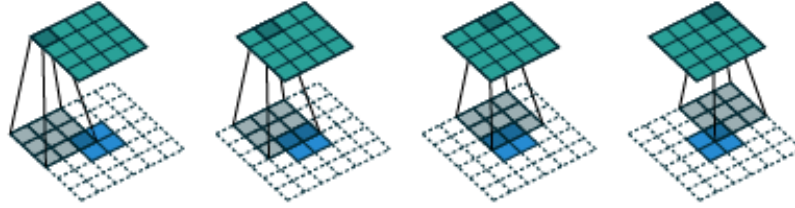


Abb. 14. Deconvolution Beispiel

2.4 Autoencoder

Autoencoder gehören zu den generativen Modellen im Bereich des Machine Learnings. Generative Modelle haben das Ziel eine Wahrscheinlichkeitsverteilungen zu erlernen. Anschließend kann diese als ein Modell genutzt werden und Samples aus dieser zu erzeugen. Die Modelle können dabei beispielsweise auf ANN oder Markov Chains trainiert werden(? , ?). Im folgenden liegt der Fokus auf ANNs. Allgemein gehalten können jegliche Typen von Daten wie Text, Bild oder Audodateien für generative Modelle herangezogen werden. Es gibt unterschiedliche Typen von generativen Modellen, welche sich vom Aufbau des Neuronalen Netzwerk und der Zielfunktion unterscheiden. Beispiele dafür sind Boltzmann Maschine, Autoencoder oder Deep Belief Networks(? , ?).

Autoencoder sind ein andere Art von Model aus den Bereich der generativen Modelle. Ihre Aufgabe besteht darin einen Input zu komprimieren und aus der komprimierten Information den Input wieder herzustellen. 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 $k \leq i$ ist. Zweitens der Decoder d parametrisiert durch θ bekommt als Input tz^k und bildet z auf x^l ab 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 Encoder es schafft das $x \approx z$. Eine grafische Darstellung kann aus Abb. 15 entnommen werden.

Die Paramter ϕ und θ werden durch durch den in Kapitel "Backprobagation Algorithmus"erlernt. Und können beispielsweise durch Fully-Connected-Layer, Convolutional-Layer oder Deconvolutional-Layer modelliert werden. Eine Metrik um zu messen wie das Model seine Aufgabe erfüllt könnte Beispielsweiße Cross-Entropy Funktionen sein welche schon in Kapitel vorgestellt worden ist.

Weitere spezifische Zielfunktionen für Autoencoder welche mit 3D-Daten arbeiten werden nun vorgestellt.

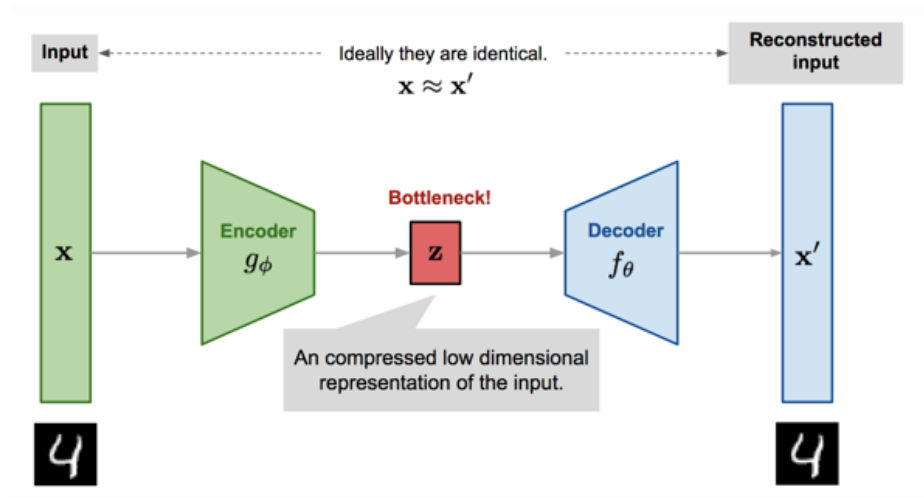


Abb. 15. autoencoder

Bei Punktwolken als Datentyp erzielt die Cross-Entropy-Zielfunktion keine gute Ergebnisse da diese Invariant zu ihrer Permutation sind(?, ?). Das heißt ändere ich die Anordnung meiner einzelnen Punkte in mein Set bleibt das Ergebnis unverändert. Deshalb kann auf die Cross-Entropy-Zielfunktion übliche Zielfunktionen welche für strukturierte Daten wie Bilder verwendet werden nicht zurück gegriffen. Das Problem dabei besteht das zwischen zwei unterschiedliche Sets von Punkten heraus zu finden wie hoch die Diskrepanz zwischen den beiden Sets ist um sie dahingehen durch Backpropagation zu optimieren diesen Abstand zu verringern. In Abbildung 7 ist dieser Prozess dargestellt. <http://graphics.stanford.edu/courses/cs468-17-spring/LectureSlides/L14>

Eine Möglichkeit speziell für Autoencoder welche mit Punktwolken arbeiten, ist die Earth Mover Distance(EMD). Bei dieser sind X^1 und X^2 zwei Punktwolken mit jeweils x^n definierten Punkten(?, ?). Definiert ist sie durch die Funktion:

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

Grafisch kann man sich die Berechnung wie in Abb. 16 darstellen. Ein Punkt von $x^n \in X^1$ wird den nächsten Punkt $y^n \in X^2$ zugewiesen. Wobei die Distanz durch die Euklidische Distanz der jeweiligen Punkte ermittelt wird.

Eine weitere Möglichkeit ist die Chamfer Distance(CD). Wie auch zuvor sind X^1 und X^2 zwei Punktwolken mit jeweils x^n definierten Punkten(?, ?). Definiert ist sie durch die Funktion:

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

Der Unterschied ergibt sich zwischen den beiden das bei der EMD von der Ausgangswolke die Punkte zu der anderen jeweils optimiert werden. Wohingegen

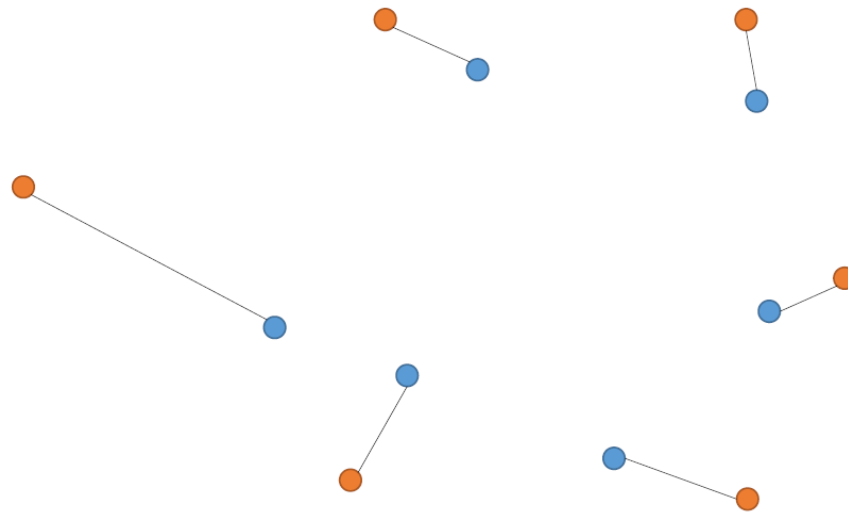


Abb. 16. Earth Mover Distance

bei der CD die Distanzen von und zu der Ausgangspunkt Wolke berechnet werden. Dies geht aus den Abb. 17 hinaus in welche die Distanzberechnung der einzelnen Punkte dargestellt wird(?, ?).

2.5 Generative Adversarial Network

In Abbildung 7 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 Model 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, das dazu dass das Integral schnell gegen unendlich konvergiert. Was dazuführt, das hohe Kosten entstehen wenn die Verteilung von generativen 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 geniert 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 abgeschlossenen Problem und wird weiterhin erforscht(?, ?). Unter GAN werden wir auf dieses Problem erneut aufgreifen und es wird gezeigt

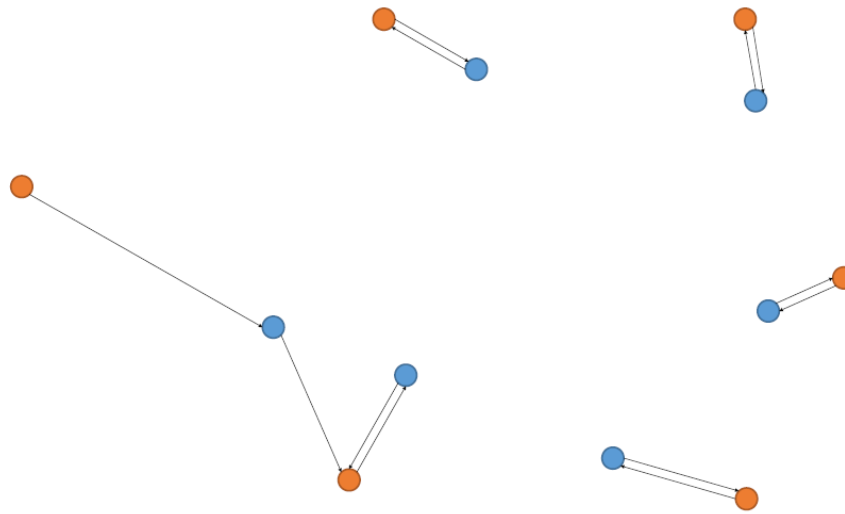


Abb. 17. Chamfer Distance

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(?, ?)

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 generierbaren 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 beobachteteten Daten am ehesten passen. Man kann ML-Schätzer als Kulback-Leibler(KL) Divergenz darstellen und das generative Modelle das Ziel haben die KL Divergenz zwischen den Trainingsdaten P_r und den generierten Daten P_g zu minimieren.

Ein GAN besteht aus zwei KNN, 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 21 entnommen werden.

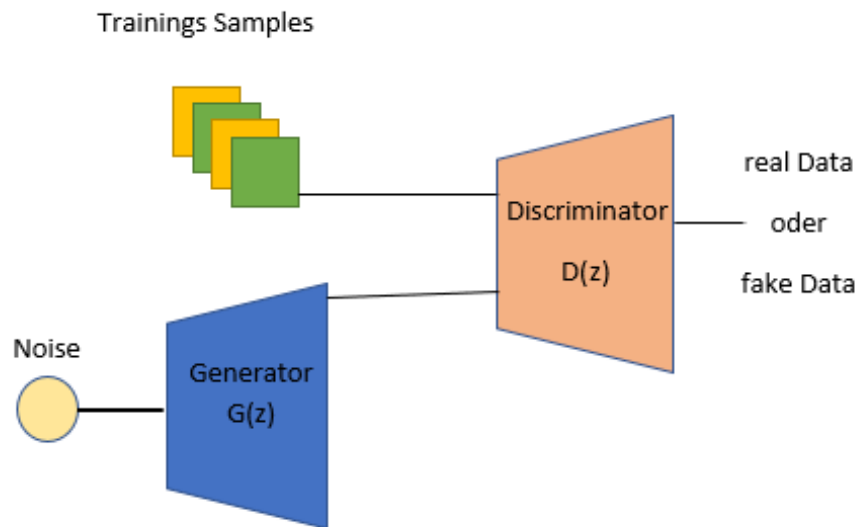


Abb. 18. Generativ Adversarial Network

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 dahingegen 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 20 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(?).

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

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

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 Ver-

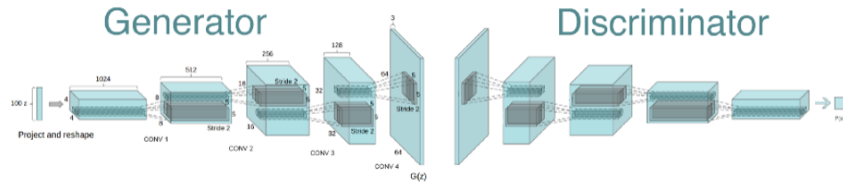


Abb. 19. Deep Convolutional GAN

gleich zu der KL Divergenz. Damit zeigte er weshalb GANs im Vorteil gegenüber anderen generativen Modellen sind. Abbildung 20 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(?).

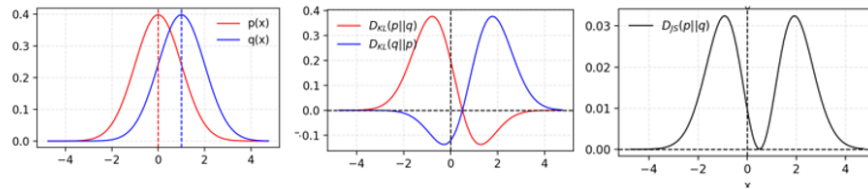


Abb. 20. KL Divergenz und JS Divergenz

2.5.1 Probleme mit Generative Adversarial Networks Wie auch anderen generativen Modelle haben auch GANs noch Schwächen bezüglich der Trainingsabläufe und der Qualität der generierten Daten. Im Folgenden wird auf einige Probleme eingegangen welche im darauffolgenden Kapitel Lösungsansätze aufgezeigt werden.

Equilibrium D und G betreiben ein MinMax Spiel. Beide versuchen das Nash Equilibrium zu finden. Dies ist der bestmögliche Endpunkt in einen 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 Maschine Learning besteht die Möglichkeit Validierungsdatensätze zu verwenden und an diesen die Genauigkeit des Modells zu testen. Diese Möglichkeit besteht bei GANs nicht (?, ?).

2.5.2 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 Gewichten 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 weist die JS immernoch Probleme auf. Es wird vorgeschlagen diese durch die Wasserstein Metric zu ersetzen, da diese bessere Ergebnisse bei disjunkten Wahrscheinlichkeitsverteilungen liefern kann(?, ?).

Wasserstein-GAN Die Wasserstein Metric oder auch Earth Mover Distance genannt misst die Minimum Kosten welche entstehen wenn man Daten von der Datenverteilung $/\alpha$ zur Datenverteilung $/\omega$ überträgt werden. Es wird oft auch von Masse oder Fläche gesprochen, welche von $/\alpha$ zu $/\omega$ getragen wird.

$$W(p_r, p_g) = \inf_{\gamma \sim \Pi(p_r, p_g)} E_{(x, y) \sim \gamma} [|x - y|]$$

p_r steht für die reale Datenverteilung zu welcher uns Daten zur Verfügung stehen und p_g steht die generierte Datenverteilung. Dabei wird nun das Infimum von allen Möglichen Transportplänen γ welche in Π enthalten sind ausgewählt. Vorteile sind unter anderem, dass der Gradient gleichmäßiger ist und das WGAN lernt besser auch wenn der Generator schlechtere Daten erzeugt im Vergleich zum üblichen GAN. Durch die Kantorovich-Rubinstein Methode kann die Wasserstein-Distanz umgeformt werden in: $W(p_r, p_\theta) = \sup_{\|f\|_L \leq 1} E_{x \sim p_r}[f(x)] - E_{x \sim p_\theta}[f(x)]$
 Durch diese Umformung ist es nun Möglich das GAN so zu gestalten durch die Funktion zu erlernen. Dabei übernimmt der Discriminator nun die Aufgabe eines Critic welcher nun nicht mehr 0 oder 1 für Fake oder Real ausgibt sondern einen Score welcher angibt wieviel Masse von der P_θ umverteilt werden muss damit der Generator bessere Ergebnisse liefert. EVTL lern algorithmus hier:

2.6 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 Generators Einfluss zu nehmen. Dabei wird das Modell so verändert das eine zusätzliche Information y als Input in den Discriminator und Generator zugefügt wird. Dabei kann y jegliche 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. 21 kann der Informationsfluss und die Konnektivität der einzelnen Module entnommen werden. Die Module Generator und Discriminator bleiben wie beim RAW-GAN und können von ihren Aufbau für die jeweiligen Datentyp verändert werden und beispielsweise durch Convolutional-Layer, Deconvolutional-Layer oder Fully-Connected-Layer bestehen.(?, ?).

2.7 3D-GAN

Das besondere an den 3D Raum im Vergleich zu Normalen 2D Bildern ist die Steigerung der Dimension und zu gleich der hohe Informationsgehalt welcher in 3D Objekten steckt. Das Ziel von 3D-GAN ist es die Datenverteilung der zugrunde liegenden 3D-Modellen zu erlernen. Dabei wird der latente Objektraum erfasst und soll dadurch die Wahrscheinlichkeiten für einzelne Objektklassen enthalten.

Es wurden schon mehrere Versuche von generativen Modellen auf 3D Daten

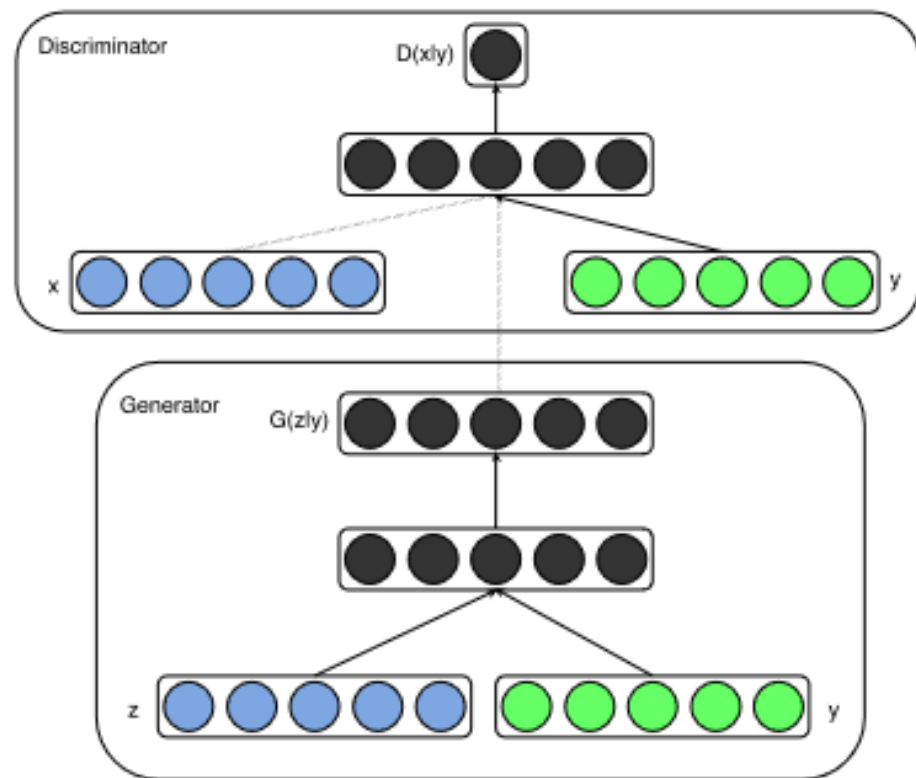


Abb. 21. Conditional Adversarial Network

durchgeführt wie von Wu, Jiajun und Zhang (?, ?) welche in ihren Model mit 3D-Voxel Daten arbeiten und damit ein GAN trainiert haben oder Achlioptas, Panos und Diamanti(?, ?) welche ein GAN auf Punktwolken trainiert. Da in folgender Arbeit die 3D-Daten durch PC dargestellt werden wird das Vorgehen von Achlioptas, Panos und Diamanti näher beleuchtet.

Die Architektur des typischen 3D-GAN in dieser Arbeit als RAW-GAN betitelt ist dem Vanilla GAN von Goodfellow ähnlich. Der Input Layer ist ein Fully-Connected Layer welcher der Anzahl der Punkte je Punktwolke $\cdot 3$ entspricht. Dieser bekommt als Input einen Noise-Vektor welcher aus einer Gausischen Verteilung entnommen wird und durch mehrere Layern gereicht bis hin zum Output Layer welche die Anzahl der gewünschten Punkte besteht. Also Zielfunktion kann mit der KL-Zielfunktion gearbeitet werden oder mit der Wasserstein Metrik welche im Versuchsaufbau von Achlioptas, Panos und Diamanti besser Ergebnisse geliefert hat. Der Aufbau unterscheidet sich nicht von Vanilla-GAN.

Eine weitere Möglichkeit ist das Latent-GAN, dieses benutzt eine andere Aufbau gegenüber dem RAW-GAN, welches dabei helfen soll den latenten Raum der Objekte zu erlernen. Zunächst wird ein Autoencoder mit den vorhandenen Trainingsdaten(siehe Autoencoder) trainiert. Der Aufbau ist der selbe einen üblichen Autoencoder beschrieben in Kapitel „Autoencoder“. Ziel dabei ist den latenten Raum der Trainingsdaten zu erlernen und eine Kompression der Daten um den Suchraum zu welcher beim RAW-GAN durch die Inputdimension gegeben ist zu verringern und dadurch das Training des GANs zu erleichtern.

Bevor das Training des Latent-GAN beginnen kann wird nun meine Trainingsdaten durch den vorher trainierten Encoder des 3D-Autoencoder auf die festgelegte Output-Dimension komprimiert. Anschließend werden diese komprimierten Daten verwendet um das GAN zu trainieren. Dabei erlernt das GAN komprimierte Code zu produzieren welche anschließend durch den Decoder des 3D-Autoencoder wieder auf die ursprüngliche Größe gebracht werden kann. Durch dieses Verfahren war es ihnen Möglich Daten in einer guten Qualität zu produzieren und eine Datenverteilung des zugrundeliegenden Models zu erlernen. Der gesamte Ablauf kann in 22 entnommen werden.

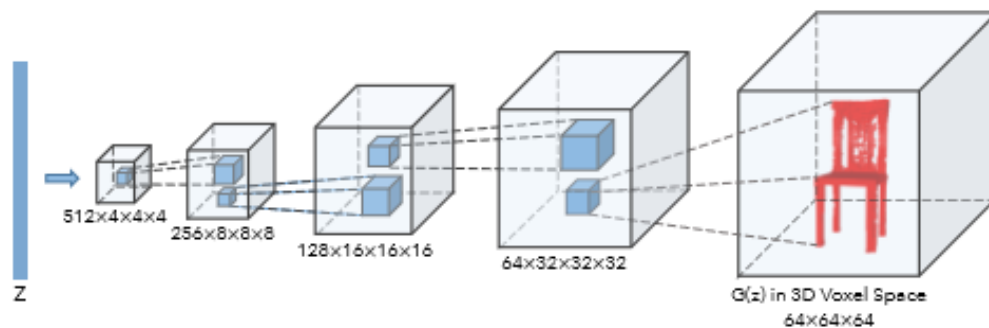


Abb. 22. Latent-GAN

(?, ?)

2.8 Rekonstruktion von Daten

Derzeit setzt Deep Learning neue Maßstäbe bei der Rekonstruktion von Daten wie Bildern oder Texte. Bei der Rekonstruktion geht es darum das Daten welche von ihren Urzustand verändert wurden, sei es durch Artefakte oder manuelle Bearbeitung wieder dahin zurück zu führen. In Abb. 23 ist eine Rekonstruktion eines Bildes dargestellt welches eine Wiederherstellung eines Hundes zeigt. Deep Learning hat in diesen Bereich besonders bei Bilder große Erfolge erlangt, dadurch das diese als Matrizen dargestellt werden können liefern sie eine Datenstruktur auf welche KNN arbeiten können. Auch durch Bearbeitung mit Convolutionen-Layer auf Bilddaten zählt als Erfolg für die Weiterverarbeitung(?, ?). Durch die genannten Methoden werden bessere Strukturen für das jeweilige Ziel, in diesem Fall der Rekonstruktion gelernt und ermöglicht es wie in verschieden Papern wie (,) welche auf super-hochauflösenden Bildern Artefakte entfernt und das Bild wieder zum Urzustand zurück führt. Ebenso wie die Arbeit von Liu, Gulin und Reda (,) welche ähnliche Ergebnisse liefern konnte. Rekonstruktion auf 3D-Daten wurde von Yi, Li und Shao (,) durchgeführt diese Arbeiteten beinhaltet jedoch die Rekonstruktion von 2D Bildern auf 3D Modellen. Bei wurde mit Hilfe von Autoencoder gearbeitet welche auf für die folgenden Arbeit von Bedeutung sind.(?, ?).



Abb. 23. Bild Rekonstruktion eines Hundes welches durch ein Artefakt zerstört wurde

3 Methoden

In diesen Kapitel werden die Methoden für die Versuchsaufbauten aufgezeigt welche die Ziele 1 - 3 überprüfen sollen. Es wird zunächst ein generelle Struktur des Aufbaus besprochen und anschließend auf spezifische gegebene eingegangen.

Außerdem werden die spezifischen Datensätze für die jeweiligen Datensätze aufgezeigt und ihre Entstehung erläutert. Alle Versuche wurden auf einem Computer mit Ubuntu 14.05 Betriebssystem mit einem Intel® Core™ i7-7700k mit 4.50GHz einer GeForce GTX 1089 mit 8G Grafikspeicher. 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 "Blätter" stammen 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 "Blätter" zurückgegriffen. Dabei wird in den Blättern Punkte herausgenommen, welche das Verdecken oder Zerstören eines Blattes simulieren soll. Dies passiert, indem eine Gauß-Verteilung auf einer 3D-Sphere erzeugt wird. Anschließend wird das Komplement zwischen einem 3D-Blatt und einer 3D-Sphere berechnet. Das Ergebnis sind zerstörte Blätter mit kreisförmigen Löchern auf der Oberfläche. Der Algorithmus zum Erzeugen der Trainingsdaten kann im Anhang entnommen werden. 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 ein AdamOptimizer, mit einem Beta1 von 0.5 und einem Beta2 von 0,5. Der Discriminator besteht aus 4-Layern, welche 1-Dimensionale Convolutional Layer bestehen. Mit einer Kernel-Größe von 1 und Stride von 1. Die Aktivierungsfunktion ist Relu. Darauf folgt 3 fully connected Layer mit der Größe 128, 54 und 1, alle mit einer Relu-Aktivierungsfunktion. Der Generator besteht aus 5 fully-connected Layern mit [64, 128, 512, 1024, 1536] Neuronen jeweils mit einer Relu-Aktivierungsfunktion. 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



Abb. 24. 3D Punktwolke einer Tabakpflanze

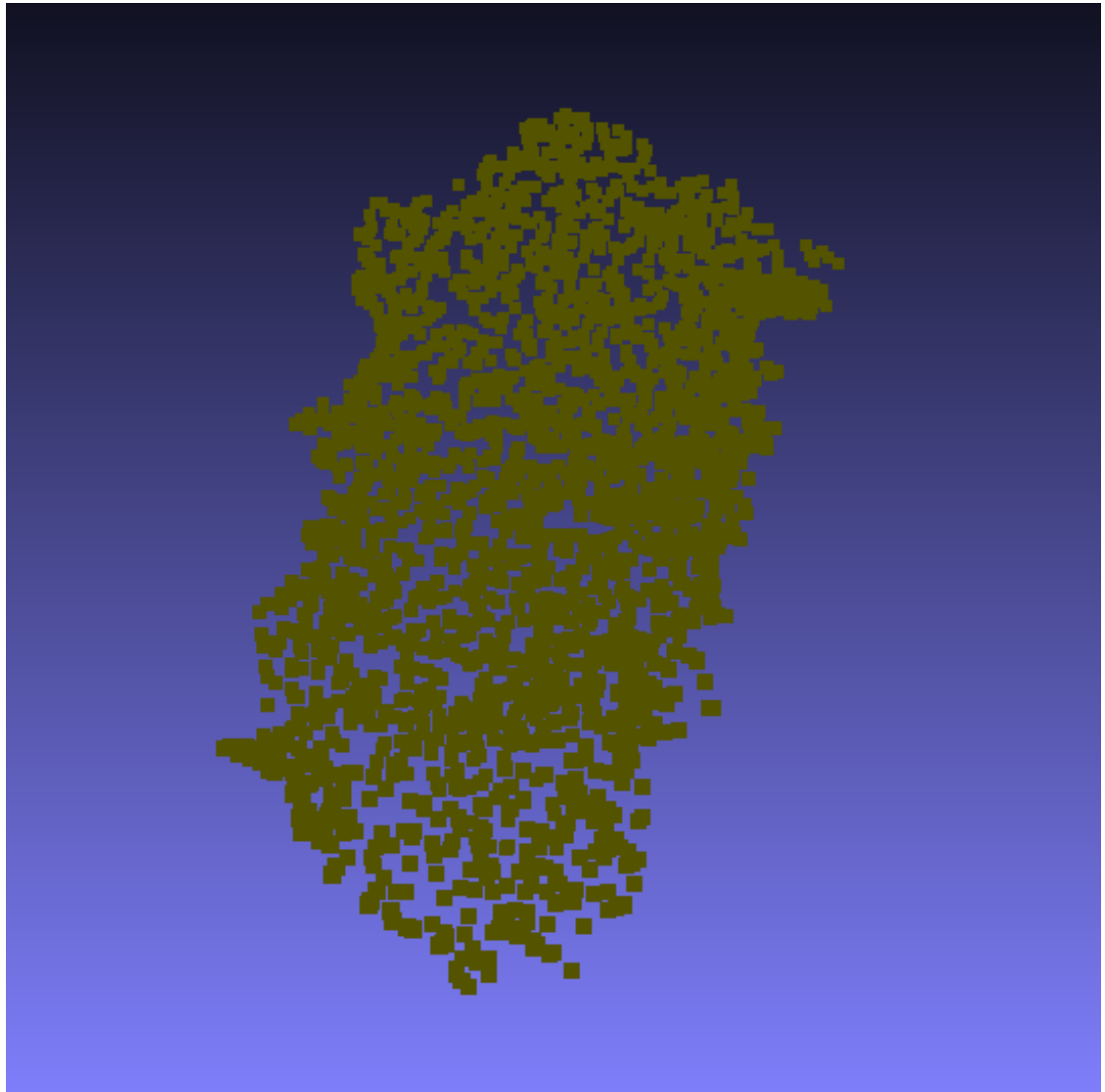


Abb. 25. 3D Punktwolke einer Tabakpflanze

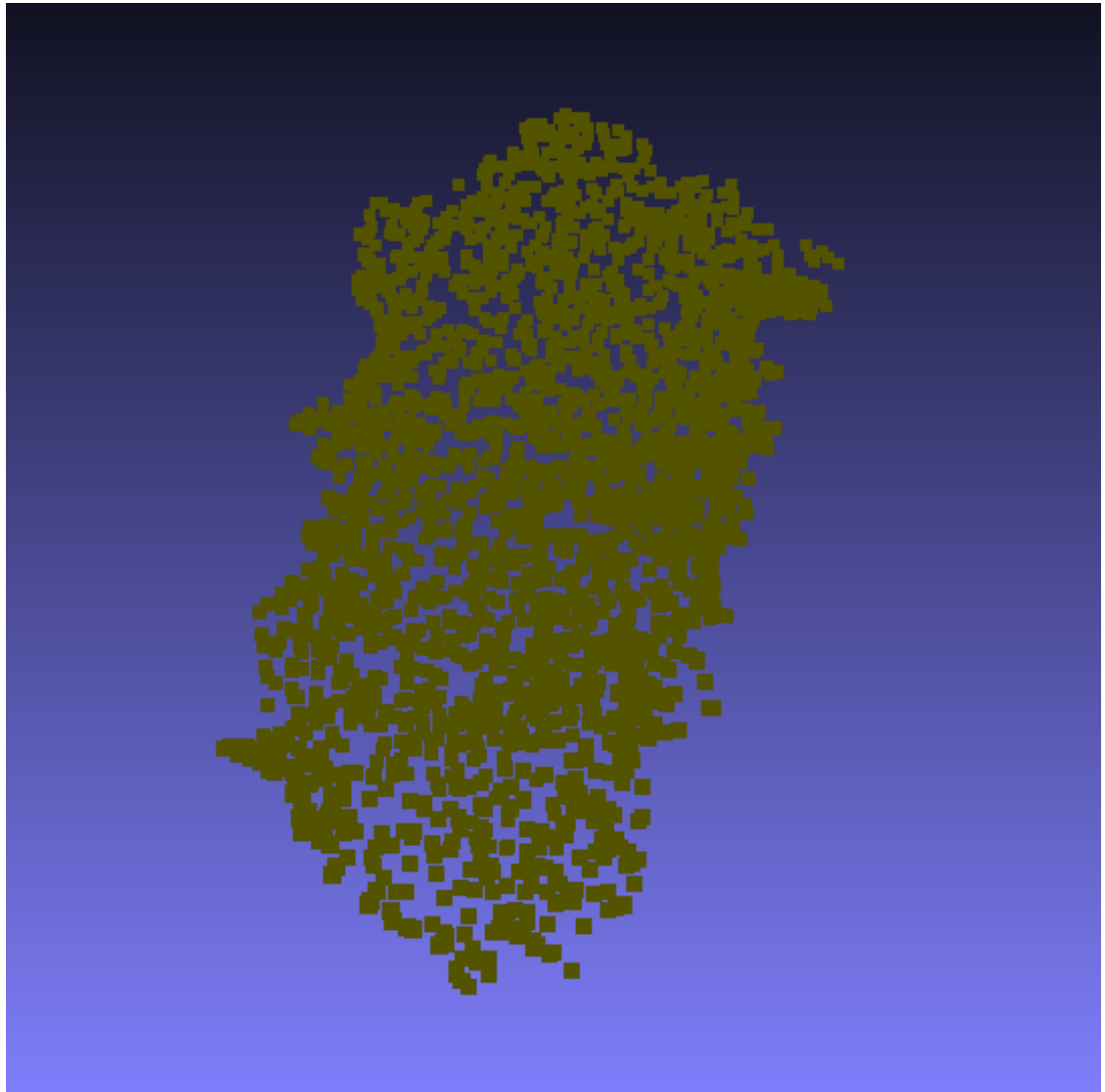


Abb. 26. 3D Punktwolke einer Tabakpflanze

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 BlattDaten x mit den trainierten Encoder zu den latenten Code y komprimiert, der einen Vektor von 128-D entspricht. Das selbe wird mit den unzerstörten BlattDaten 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 beschrieben C-GAN üblichen Aufbau. der Generator bekommt als Input x' und x welches einen 128-D Vektor welcher aus einer Gausischen Verteilung gesampelt 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.

4 Evaluation und Ergebnisse

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.

4.1 Ergebnisse - Versuchsaufbau 1

Die Ergebnisse von RAW-GAN mit den BlattDaten welches nach 500 Epochen Training lassen sich in Abb. entnehmen. Wie zu sehen ist sind die Blätter nicht

gut genug für eine Weiterverarbeitung. Die Daten erinnern eher an einen hohlen Ball als an Blättern. Auch lässt der Trainingsverlauf aus Abb. zu entnehmen darauf schließen dass keine weitere Verbesserung möglich ist auch auf längere Epochen gesehen da es zu viele Sprünge in der Zielfunktion den Discriminator zu gut werden lässt und den Generator nicht verbessern lässt.

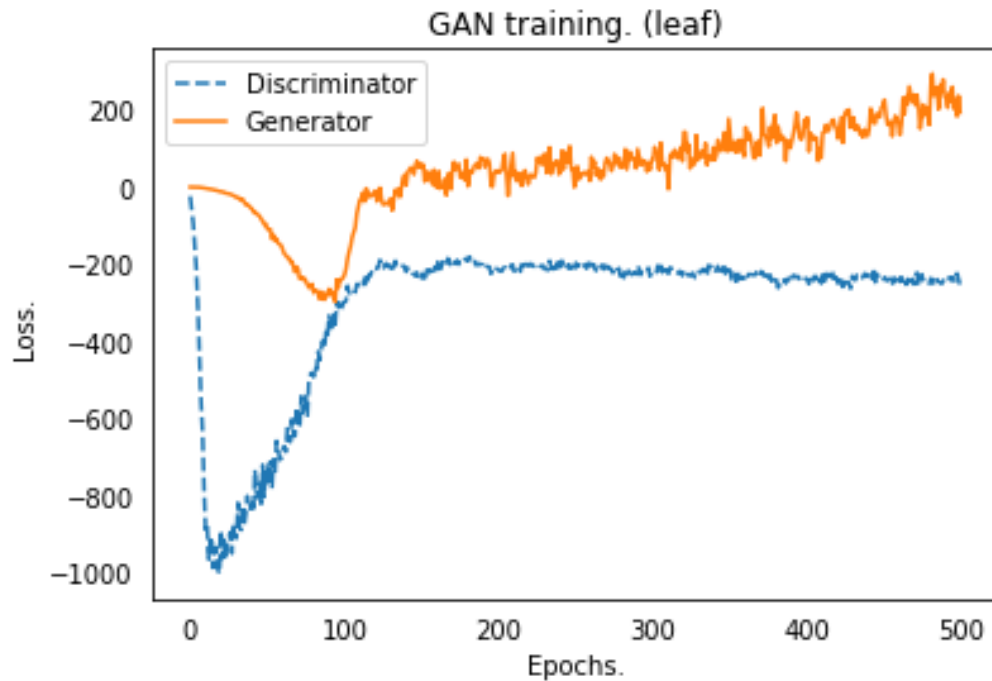


Abb. 27. Trainingsverlauf des RAW-GAN mit den Blattdaten

Ähnliche Ergebnisse sind auch mit den Stuhl Daten zu beobachten in Abb. jeweils die Ergebnisse dafür. Es sind besser die Strukturen zu erkennen als bei den Blättern ebenfalls nicht optimale Ergebnisse. In Abb. der Trainingsverlauf.

Latent-GAN chair Latent-GAN leaf

4.2 Ergebnisse - Versuchsaufbau 2

Die Ergebnisse für das Latent-GAN mit den Blatt Daten welches zunächst für 500 Epochs mit den Autoencoder trainiert wurde können Input und Output Paare aus Abbildung entnommen werden in Abbildung sind die Loss jeweils mit EMD-Loss und Damfer Distanz aufgelistet.

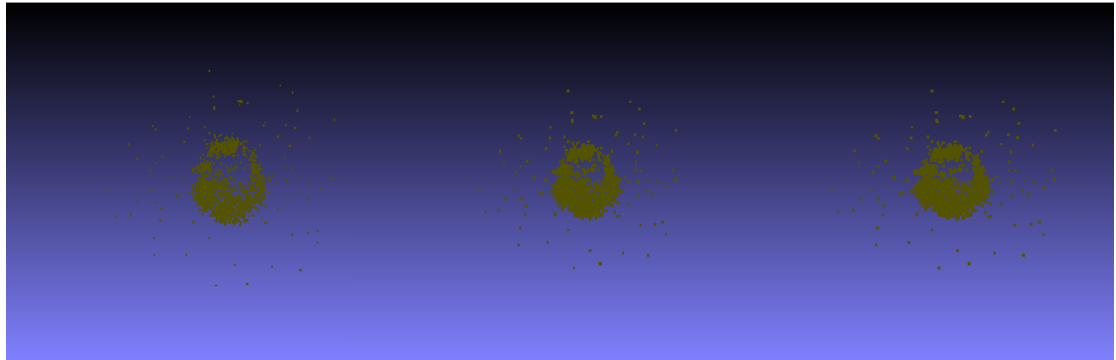


Abb. 28. Trainingsverlauf des RAW-GAN mit den Blattdaten

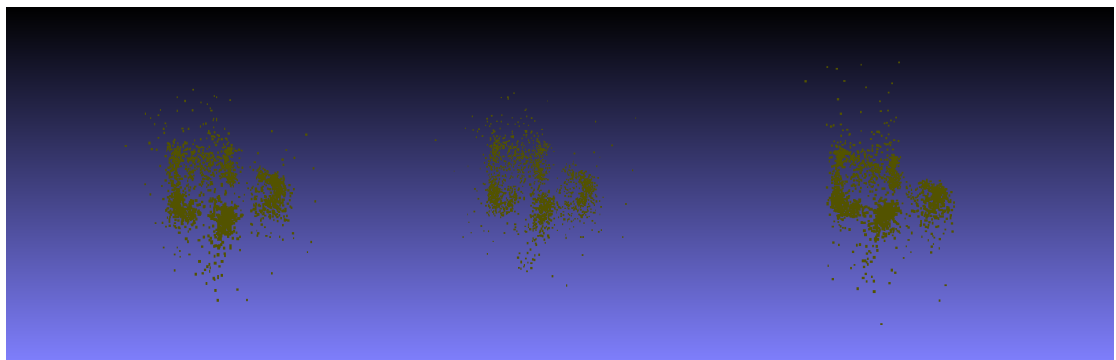


Abb. 29. Trainingsverlauf des RAW-GAN mit den Stuhl原因 mit 422 Trainingsdaten

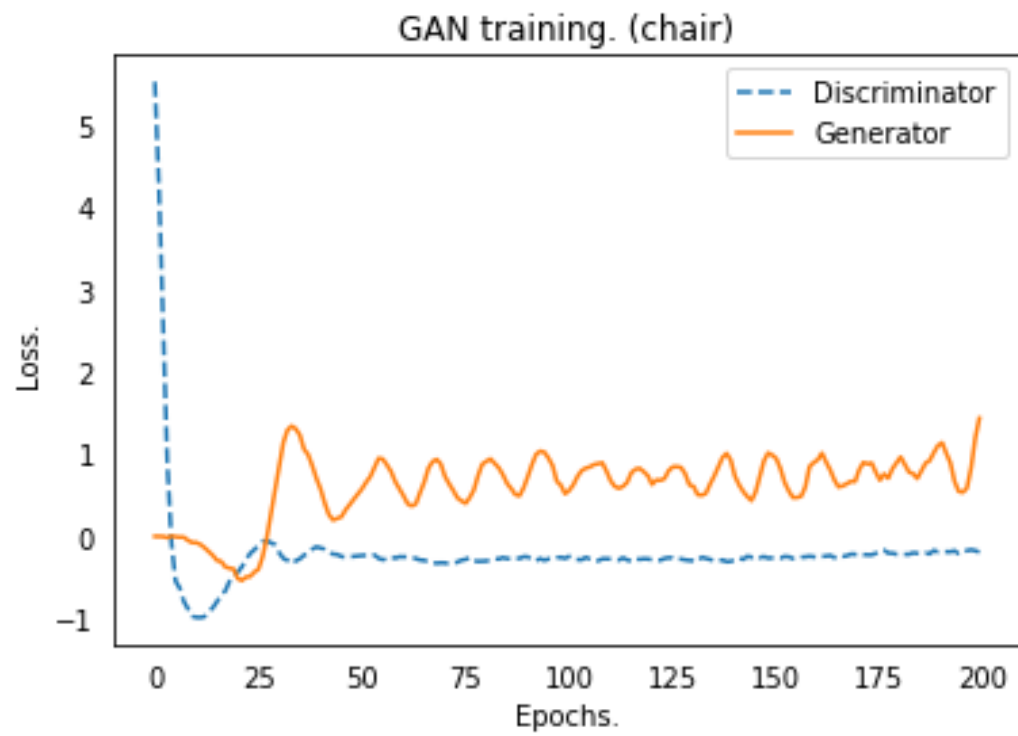


Abb. 30. Trainingsverlauf des RAW-GAN mit den Stuhl­daten mit 422 Trainingsdaten

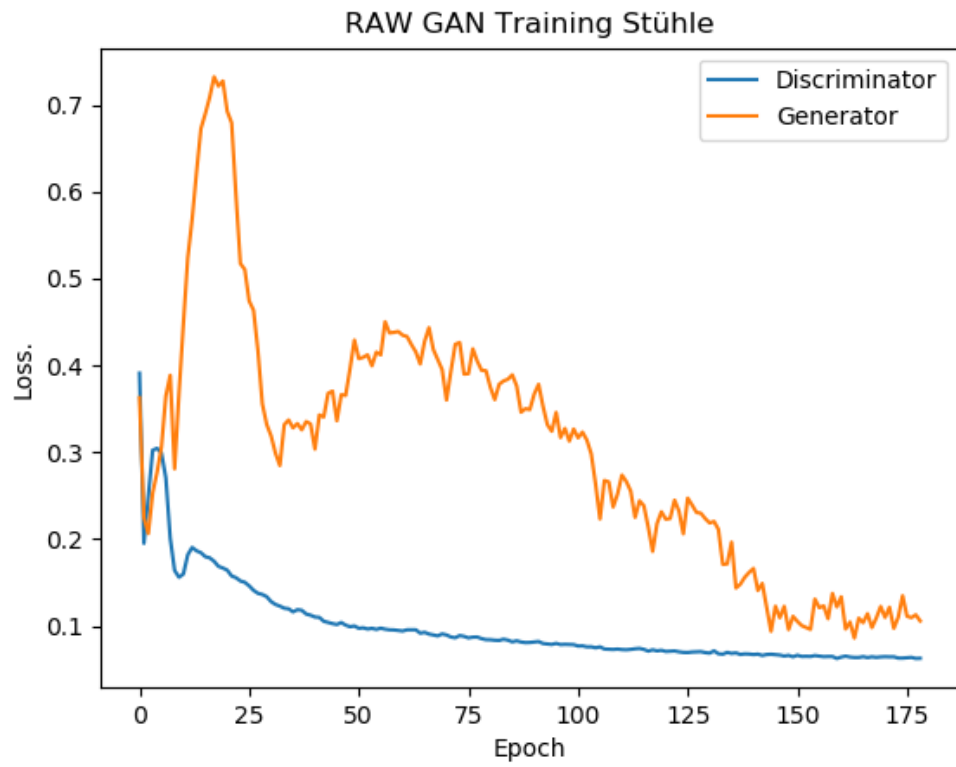


Abb. 31. Trainingsverlauf des RAW-GAN mit den Stuhlzeiten

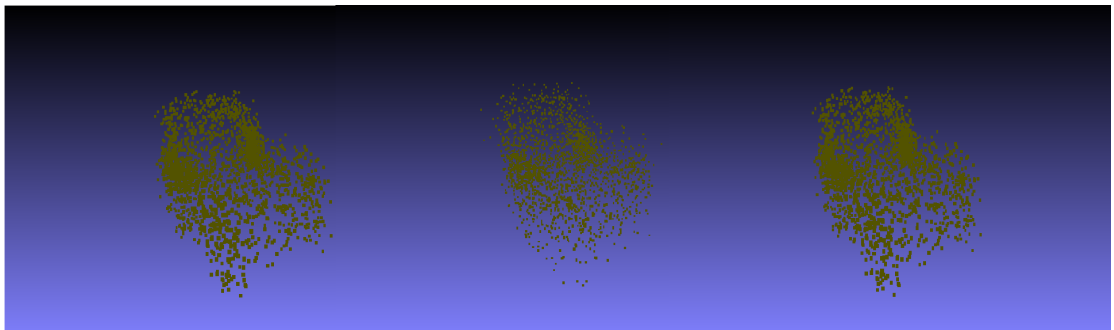


Abb. 32. Trainingsverlauf des RAW-GAN mit den Stuhlzeiten

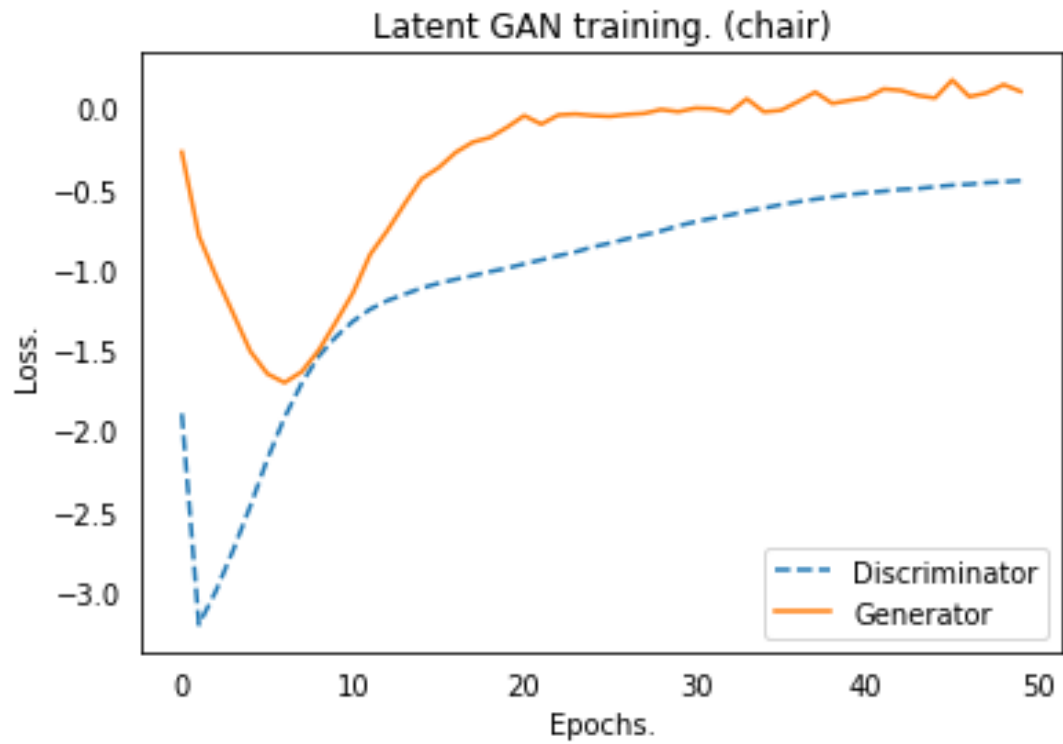


Abb. 33. Trainingsverlauf des Latent-GAN mit den Stuhldaten

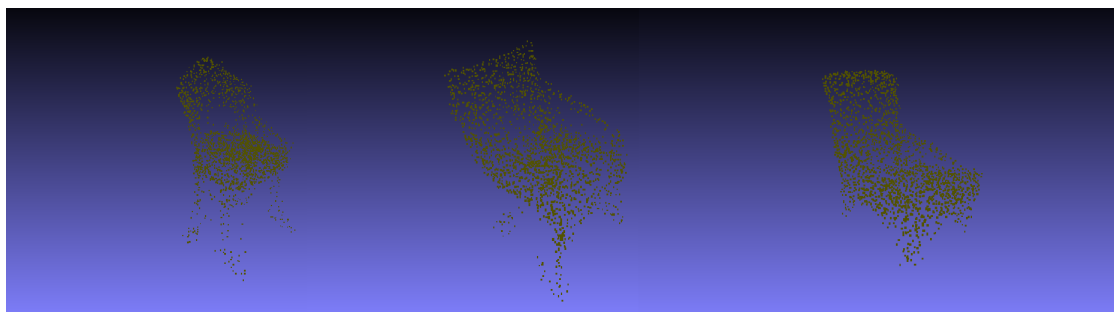


Abb. 34. Trainingsverlauf des Latent-GAN mit den Stuhldaten

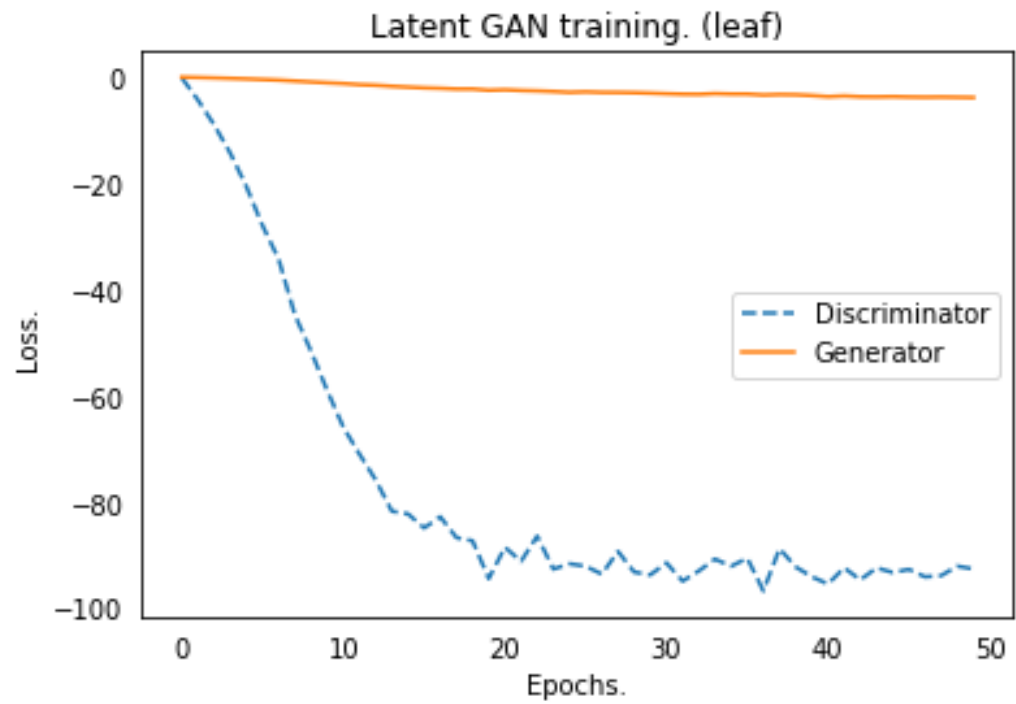


Abb. 35. Trainingsverlauf des Latent-GAN mit den Blatttdaten

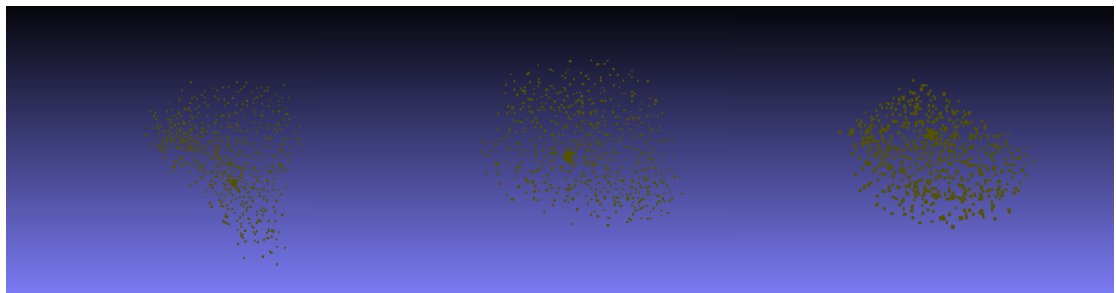


Abb. 36. Trainingsverlauf des Latent-GAN mit den Blatttdaten

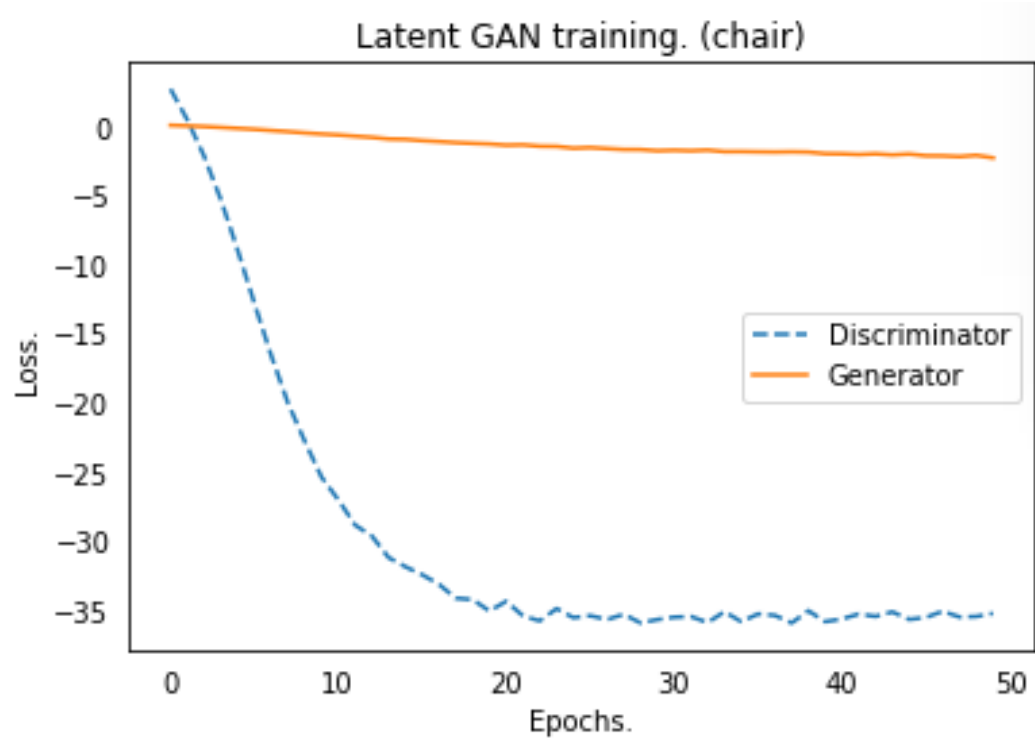


Abb. 37. Trainingsverlauf des Latent-GAN mit den Stuhldaten und 400 Trainingsdaten

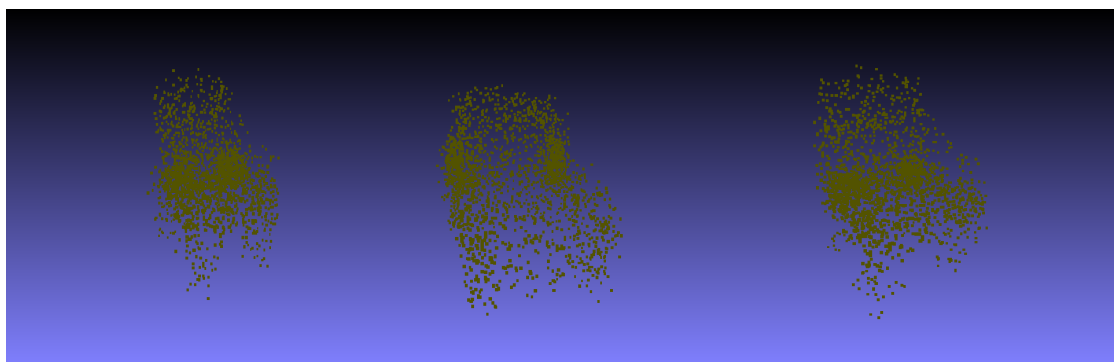


Abb. 38. Trainingsverlauf des Latent-GAN mit den Stuhldaten und 400 Trainingsdaten

Die Ergebnisse für das Latent-GAN mit den Tischdaten welches zunächst für 500 Epochs mit den Autoencoder trainiert wurde können Input und Output Paare aus Abbildung entnommen werden in Abbildung sind die Loss jeweils mit EMD-Loss und Damfer Distanz aufgelistet.

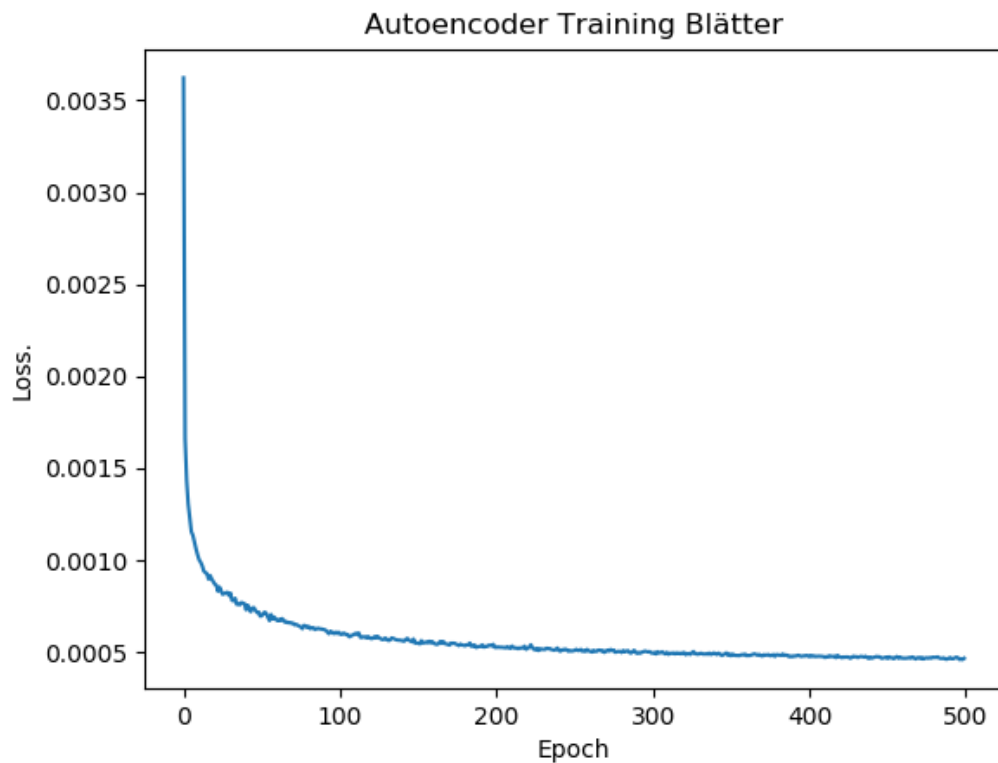


Abb. 39. Trainingsverlauf des RAW-GAN mit den Blattdaten

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

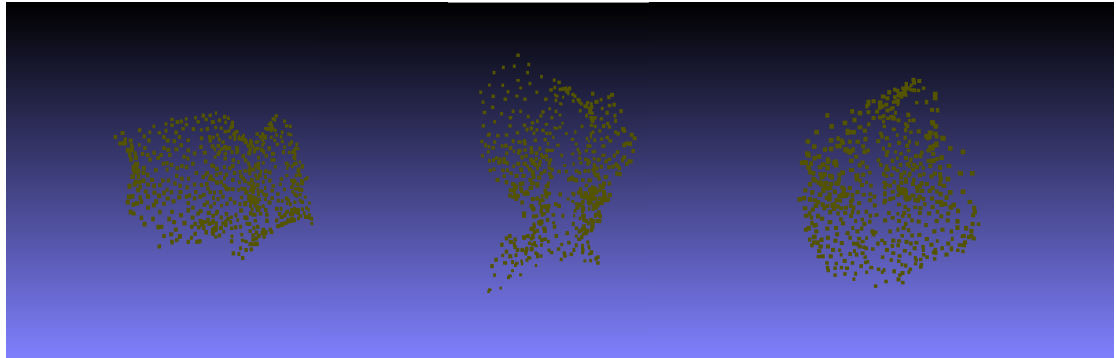


Abb. 40. Trainingsverlauf des Autoencoders mit den zerstörten Blattdaten CD Output

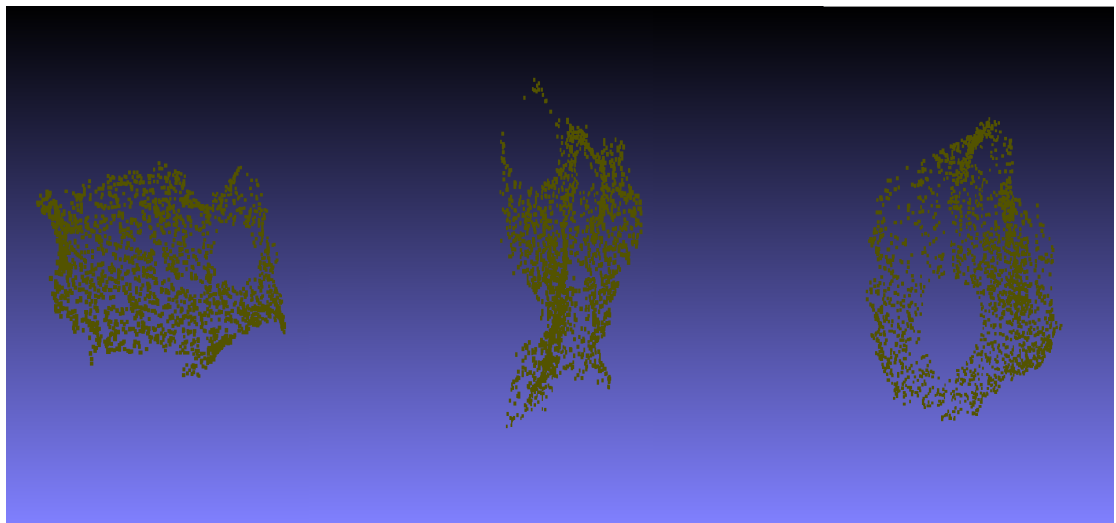


Abb. 41. Trainingsverlauf des Autoencoders mit den zerstörten Blattdaten CD input

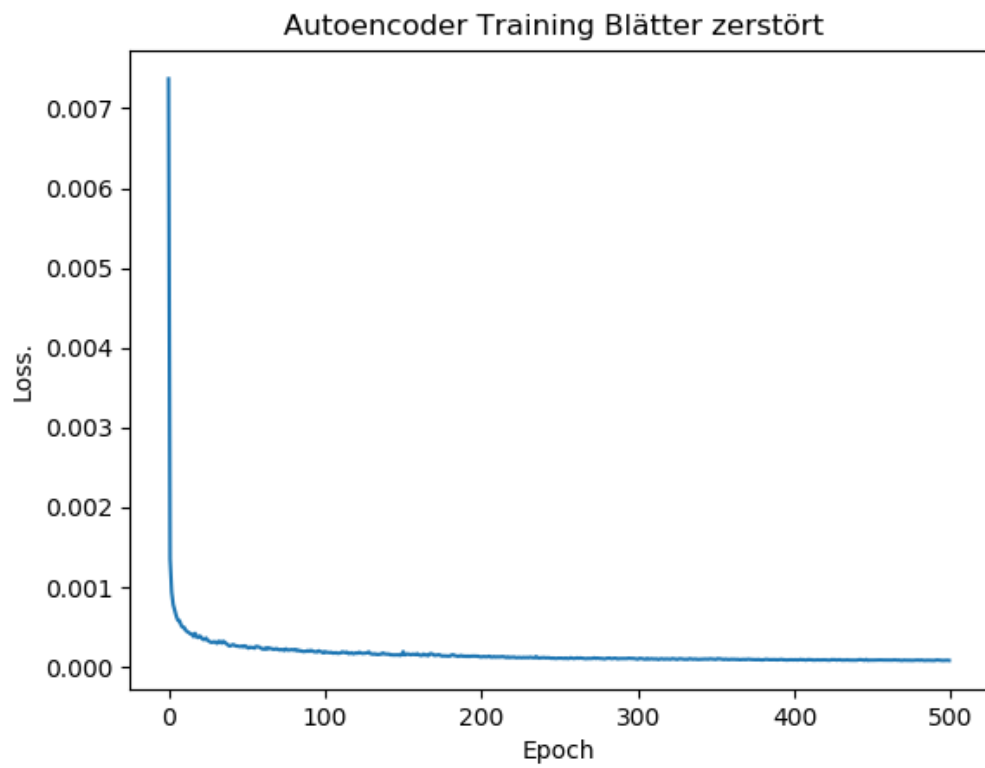


Abb. 42. Trainingsverlauf des Autoencoders mit den zerstörten Blattdaten CD

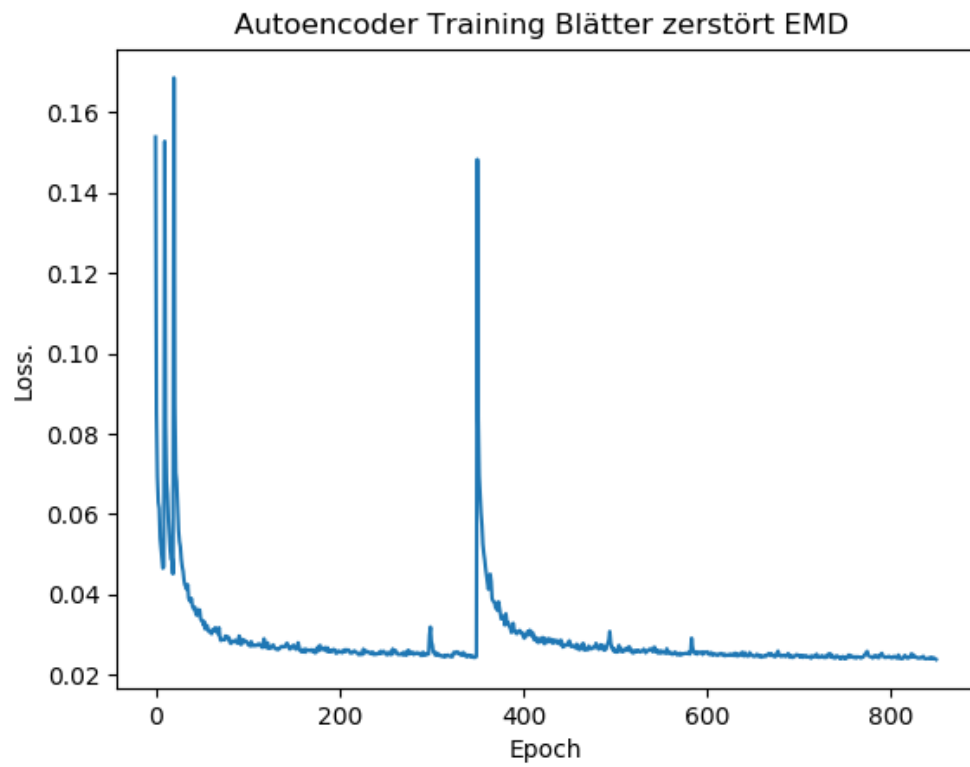


Abb. 43. Trainingsverlauf des Autoencoders mit den zerstörten Blatttdaten EMD

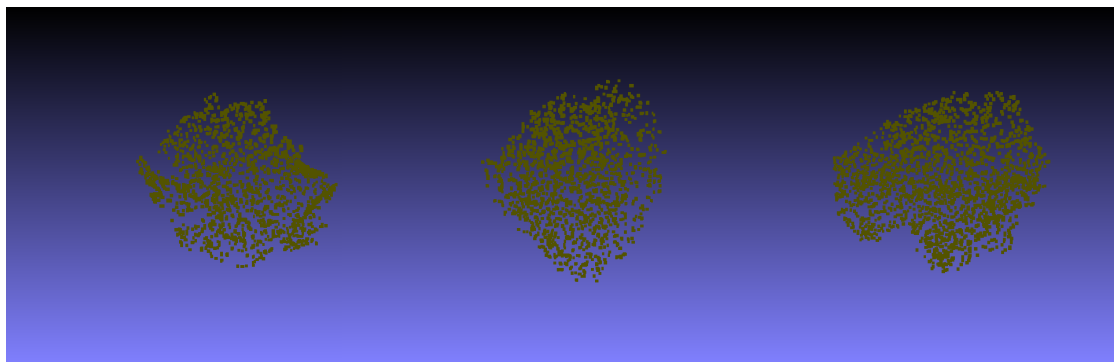


Abb. 44. Trainingsbeispiele des Autoencoders mit den zerstörten Blatttdaten EMD

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 "versteckteRepräsentation 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	9
5	Minimum Maximum Saddle Point	10
6	LossFunktion	10
7	aaaa	13
8	Punktwolke eines Tabakblattes	15
9	Polygon File Format	16
10	TERRA-REF Feld Scanner	16
11	Scankopf	17
12	Convolutional Neural Network	18
13	Convolution Beispiel	19
14	Deconvolution Beispiel	20
15	autoencoder	21
16	Earth Mover Distance	22
17	Chamfer Distance	23
18	Generativ Adversarial Network	24
19	Deep Convolutional GAN	26
20	KL Divergenz und JS Divergenz	26
21	Conditional Adversarial Network	30
22	Latent-GAN	31
23	Bild Rekonstruktion eines Hundes welches durch ein Artefakt zerstört wurde	32
24	3D Punktwolke einer Tabakpflanze	34
25	3D Punktwolke einer Tabakpflanze	35
26	3D Punktwolke einer Tabakpflanze	36
27	Trainingsverlauf des RAW-GAN mit den Blattdaten	38
28	Trainingsverlauf des RAW-GAN mit den Blattdaten	39
29	Trainingsverlauf des RAW-GAN mit den Stuhlzeiten mit 422 Trainingsdaten	39
30	Trainingsverlauf des RAW-GAN mit den Stuhlzeiten mit 422 Trainingsdaten	40
31	Trainingsverlauf des RAW-GAN mit den Stuhlzeiten	41
32	Trainingsverlauf des RAW-GAN mit den Stuhlzeiten	41
33	Trainingsverlauf des Latent-GAN mit den Stuhlzeiten	42
34	Trainingsverlauf des Latent-GAN mit den Stuhlzeiten	42
35	Trainingsverlauf des Latent-GAN mit den Blattzeiten	43
36	Trainingsverlauf des Latent-GAN mit den Blattzeiten	43
37	Trainingsverlauf des Latent-GAN mit den Stuhlzeiten und 400 Trainingsdaten	44

38	Trainingsverlauf des Latent-GAN mit den Stuhl- und 400	
	Trainingsdaten	44
39	Trainingsverlauf des RAW-GAN mit den Blatt- und 45	
40	Trainingsverlauf des Autoencoders mit den zerstörten Blatt- und 46	
	CD Output	46
41	Trainingsverlauf des Autoencoders mit den zerstörten Blatt- und 46	
	CD input	46
42	Trainingsverlauf des Autoencoders mit den zerstörten Blatt- und 47	
43	Trainingsverlauf des Autoencoders mit den zerstörten Blatt- und 48	
44	Trainingsbeispiele des Autoencoders mit den zerstörten Blatt- und 48	
	EMD	48

Tabellenverzeichnis