

# Masterarbeit

von cand. mecha. Lukas Harsch

## Vergleich von Methoden für das semi-überwachte Lernen mit neuronalen Netzen

Abgabetermin: 30.07.2018

Ausgegeben von

Prof. Dr.-Ing. Stefan Riedelbauch

INSTITUT FÜR STRÖMUNGSMECHANIK UND HYDRAULISCHE  
STRÖMUNGSMASCHINEN (IHS)  
Fakultät Energie-, Verfahrens- und Biotechnik  
Universität Stuttgart

Betreut von

M.Sc. Andreas Look

und durchgeführt am

INSTITUT FÜR STRÖMUNGSMECHANIK UND HYDRAULISCHE  
STRÖMUNGSMASCHINEN (IHS)  
Fakultät Energie-, Verfahrens- und Biotechnik  
Universität Stuttgart

# Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit nur mit den angegebenen Hilfsmitteln, und ohne unerlaubte Hilfe selbstständig angefertigt und verfasst habe.

Stuttgart, den xxx

Vorname Nachname

# Kurzzusammenfassung

Hier kommt die Kurzzusammenfassung (Abstrakt) auf Deutsch herein. Maximal eine Seite.

# Inhaltsverzeichnis

Nomenklatur	iv
<b>1 Einleitung</b>	<b>1</b>
<b>2 Grundlagen</b>	<b>2</b>
2.1 NN basics . . . . .	2
2.2 Neuronale Netze . . . . .	6
2.3 Semi-überwachtes Lernen . . . . .	7
2.3.1 Generative Netze . . . . .	8
2.3.2 Attack-Defence . . . . .	11
2.4 Akustic-Preprocessing . . . . .	11
<b>3 Ergebnisse</b>	<b>12</b>
3.1 Autoencoder . . . . .	12
3.2 GAN . . . . .	12
3.3 Attak-Defence . . . . .	12
<b>4 Zusammenfassung</b>	<b>13</b>
Literaturverzeichnis	14
<b>A Inhaltliche Ergänzungen</b>	<b>15</b>

# Nomenklatur

## Lateinische Buchstaben

$g$	$\text{m/s}^2$	Erdbeschleunigung
$H$	m	(Fall)Höhe

## Griechische Buchstaben

$\Delta$	%	Abweichung
$\epsilon$	$\text{m}^2/\text{s}^3$	isotrope Dissipationsrate

## Tiefgestellte Indizes

1,2	Eintritt, Austritt
abs	absolut
hyd	hydraulisch

## Abkürzungen

BP	Betriebspunkt
DT	(Teil)Rechengebiet Saugrohr

# 1 Einleitung

Das folgende Kapitel gibt für die häufigsten und gebräuchlichsten Befehle eine kleine Anleitung, wie beispielsweise das Einfügen eines Bildes, einer Formel oder einer Fußnote.

Das Kapitel ?? beschreibt das übliche Vorgehen bei einer wissenschaftlichen Arbeit und dient als Hilfe. Kapitel ?? listet einige nützliche Programme auf.

## 2 Grundlagen

TODO

### 2.1 NN basics

Intro: - NN approximiert  $f^*$

- $y = f^*(x)$  mapping von Input  $x$  auf Klasse  $y$
- $y = f(x, \theta)$  learns parameter of the NN
- $f^{(1)}, f^{(2)}$  und  $f^{(3)}$  zu  $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$  gerichteter Graph
- $f^{(i)}$  einzelnen Ebenen des NN
- Input-Layer nimmt eingangsdaten auf
- Ausgangsebene soll Ausgabe erzeugen, möglichst ähnlich zu  $y$
- Hidden layers bestehen aus vielen parallel arbeitenden einheiten, wobei jede eine vektor-zu-skalar funktion repräsentiert. (Neuron nimmt viele werte von vorherigen Einheiten als Input und berechnet eigene Aktivierung)

Cost funktion:

- wichtiger punkt ist die Kostenfunktion in der NN architektur
- ... irgendwas mit: die Kostenfunktion gibt vor welche Aufgabe das Netz lernen soll.
- in vielen Fällen definiert das Model eine Verteilung  $p(y|x; \theta)$ . So kann das Netz mit dem principle of maximum likelihood trainiert werden. Das bedeutet die Kostenfunktion ist definiert als die negative log-Wahrscheinlichkeit. Dies wird auch bezeichnet als die cross-entropy zwischen den Trainingsdaten und den model predictions. Formal wird die Kostenfunktion beschrieben als:

$$J(\theta) = -E[\log p_{\text{model}}(\mathbf{y}|\mathbf{x})]. \quad (2.1)$$

- Durch die Spezialisierung der Kostenfunktion kann das Netz so trainiert werden dass es andere Schätzungen durchführt.
- Oft setzt sich die gesamte Kostenfunktion aus einer primären Kostenfunktion und einem Regularisierungsterm zusammen.

Output units:

- Output units eng gekoppelt mit den Output units, da die Repräsentation des Outputs die Form der Kostenfunktion beeinflusst/vorgibt.
- im Prinzip können diese Units überall im Netz verwendet werden. Hier liegt der Fokus aber auf ihrer Verwendung als Output des Models.
- Netzwerk stellt ein Set aus hidden features bereit, definiert durch  $\mathbf{h} = f(\mathbf{x}; \theta)$ . Die

Output units wenden eine zusätzliche Transformation auf die feature  $\mathbf{h}$  an um so die Ausgabe des Netzwerkes zu vervollständigen, entsprechend der definierten Aufgabe des Netzes.

Linear:

- Die einfachste Form ist eine lineare Unit. Diese wendet eine affine Transformation ohne Nichtlinearität an. Eine lineare output unit produziert einen Vektort  $\hat{y} = W^t h + b$  für gegebene Features  $\mathbf{h}$ . lineare output units werden oft verwendet um Erwartungswert einer bedingten Gaussverteilung zu schätzen:

$$p(y|x) = N(y; \hat{y}, I). \quad (2.2)$$

- Maximizing the log-likelihood is then equivalent to minimizing the mean squared error. (in the linear case)

Sigmoid:

- Viele Aufgaben benötigen die Schätzung einer binären Variable  $y$ , z.B. die Klassifikation mit zwei Klassen.

- Dieses Problem wird als Bernoulliverteilung über  $y$  bedingt durch  $x$  modelliert. Die Bernoulliverteilung ist definiert als...(mglw hier einfügen oder weg lassen). Somit muss das Netz nur  $P(y = 1|x)$  schätzen.

- Um diese Bedingung zu erfüllen wird eine sigmoid unit verwendet.

- Dazu wird die sigmoid unit um Probleme in dieser Form zu beschreiben.

- Die sigmoid output unit ist definiert als:  $\hat{y} = \sigma(w^T h + b)$ . Dabei ist  $\sigma$  die logistic sigmoid function.

—  
- (3.10) logistic sigmoid:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (2.3)$$

- die logistic sigmoid funktion wird im Allgemeinen verwendet um die Parameter  $\phi$  einer Bernoulliverteilung zu ermitteln, da die Funktion im Bereich (0,1) definiert ist. Somit liegt die Funktion im Wertebereich des Parameter  $\phi$  der Bernoulliverteilung.

- die Sigmoidfunktion ist gesättigt falls ihr Argument sehr positiv oder sehr negativ ist. Somit ist die Funktion sehr flach und unempfindlich gegenüber kleinen Änderungen im Eingang.

—  
- Zuerst lineare layer  $z = w^T h + b$ . Danach sigmoid aktivierung funktion um  $z$  in eine Wahrscheinlichkeit zu konvertieren.

Softmax:

Die softmax Funktion kann verwendet werden um eine Wahrscheinlichkeitsverteilung über eine diskrete Variable mit  $n$  möglichen Werten zu beschreiben. Im Grunde eine Verallgemeinerung der sigmoid funktion. Meistens wird die softmax Funktion als



Ausgang einer Klassifikation verwendet um die Wahrscheinlichkeitsverteilung über  $n$  verschiedenen Klassen zu repräsentieren.

- generalisierung der sigmoid funktion zur schätzung der bernoulliverteilung. ergibt  $\hat{y}_i = P(y = i|x)$ . Dabei soll nicht nur jedes Element von  $\hat{y}_i$  zwischen 0 und 1 liegen sondern auch die Summe des gesamten Vektors = 1 sein.
- Somit gilt der Ansatz allgemein für multinoulli verteilungen
- Wie zuvor wird zuerst eine lineare Layer  $z = w^T h + b$  angewendet. Anschließend wird die softmax aktivierung angewendet, beschreiben durch:  $\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$

- vltt beispiel aus Präsi wie beliebige Werte über die softmax in Verteilung umgerechnet werden + Bild von MNIST mit zugehörigem Barendiagramm für die Verteilung

Hidden layer:

- Im allgemeinen werden die hidden units als Funktion beschreiben welche auf einen Vektor aus Eingangsdaten  $\mathbf{x}$  eine affine Transformation  $z = W^T x + b$  durchführt. Dabei beschreibt  $W$  eine Gewichtsmatrix und  $b$  einen Bias-Vektor. Anschließend wird eine elementweise nichtlineare Funktion  $g(z)$  angewendet. Die Funktion  $g(z)$  wird Aktivierungsfunktion genannt.
- hidden unit für jede Ebene werden somit durch  $h^{(l)} = g(W^{(l)T} x + b^{(l)})$  beschrieben, wobei  $l$  die jeweilige Ebene vorgibt.
- Die meisten hidden units unterscheiden sich durch eine unterschiedliche Wahl der Aktivierungsfunktion  $g(z)$ .
- häufig werden rectified linear units (ReLU) verwendet mit der aktivierungsfunktion:  $g(z) = \max\{0, z\}$ .
- Diese verhalten sich wie eine lineare Aktivierung nur dass die Outputs über die linke Hälfte der Domäne =0 sind.
- Relun wird typischerweise ontop einer affinen Transformation angewandt:  $h = g(W^T x + b)$ .

- Erweiterung ist die leaky ReLU  $f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{otherwise} \end{cases}$
- Darüberhinaus existieren viele weitere Aktivierungsfunktionen.

Conv networks:

- Erweiterung der bisher beschriebenen NN sind die convolutional neural networks (CNN).
- Großen Erfolg in der Bildverarbeitung, bei welchem der Input als 2D grid von pixeln betrachtet werden kann.
- wie der Name sagt wird in diesen Netzen eine convolutional Operator eingeführt.

Conv operator:

- Convolution im Allgemeinen ist eine Operation zwischen zwei Funktionen  $x$  und  $w$ .

Dabei wird das Integral der punktweisen Multiplikation der zwei Funktionen gebildet, wobei eine Funktion gedreht und verschoben ist. - Dies ist definiert als:

$$s(t) = (x \star w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a). \quad (2.4)$$

- Bei der Anwendung von CNNs ist der Eingang typischerweise ein multidimensionaler Array aus Daten. Der Kernel beschreibt ein multidimensionalen Array aus Parameter die während des Trainings vom Lernalgorithmus angepasst werden.

- oftmals wird convolution über mehr als eine Achse berechnet. Für ein zweidimensionales Bild  $I$  als Input beispielsweise kann ein zweidimensionaler Kernel  $K$  verwendet werden. Die Convolution resultiert in:

$$S(i, j) = (I \star K)(i, j) = \sum_m \sum_n I(m, n)K(i-m, j-n). \quad (2.5)$$

Bild von Faltungsprozess und Referenze einfügen.

### Motivation

- CNNs haben drei Ideen welche helfen das machine learning Verhalten zu verbessern: sparse interactions, parameter sharing and equivariant representations.

- Traditionelle NN Ebenen verwenden eine Matrix multiplikation zwischen einer Matrix aus Parametern und einem weiteren Parameter. Das bedeutet jede Output unit interagiert mit jeder Eingangs unit. CNN dagegen haben eine **sparse interaction**. Dies wird erreicht indem kleiner als der Input ist. Ein Bild beispielsweise besteht aus einer Vielzahl von Pixeln. Ein kleinerer Kernel betrachtet nur wenige Pixel aus dem gesamten Bild und kann somit aussagekräftige Features wie Kanten und Ecken in dem Bild erkennen. Darüber hinaus werden weniger Parameter benötigt wodurch der Speicher- und Rechenaufwand reduziert wird.

- **parameter sharing**: In traditionellen NN wird jedes Element der Gewichtsmatrix genau ein mal verwendet um den Ausgang einer Ebene zu berechnen, indem jedes Gewicht mit einem Element des Inputs multipliziert wird und danach nicht mehr verwendet wird. In einem CNN dagegen wird jeder Filter eines Kernels an jeder Position des Inputs verwendet.

- The parameter sharing used by the convolution operation means that rather than learning a separate set of parameters for every location, we learn only one set

- Des Weiteren haben CNN die Eigenschaft **equivariance to translation**. Für eine Funktion bedeutet dies, wenn sich der Input ändert, so ändert sich der Output gleichermaßen. Formel geschrieben ist eine Funktion  $f(x)$  equivariant zu einer Funktion  $g$  falls  $f(g(x)) = g(f(x))$  ist.

- When processing time series data, this means that convolution produces a sort of timeline that shows when different features appear in the input. If we move an event later in time in the input, the exact same representation of it will appear in the output, just later in time.

goodfellow S.350

- nahe zu immer in verbildung mit convs ist die pooling operation.

## Pooling

## 2.2 Neuronale Netze

- NN approximiert  $f^*$
- $y = f^*(x)$  mapping von Input  $x$  auf Klasse  $y$
- $y = f(x, \theta)$  learns parameter of the NN
- $f^{(1)}$ ,  $f^{(2)}$  und  $f^{(3)}$  zu  $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$  gerichteter Graph
- $f^{(i)}$  einzelnen Ebenen des NN
- Input-Layer nimmt eingangsdaten auf
- Ausgangsebene soll Ausgabe erzeugen, möglichst ähnlich zu  $y$
- Hidden layers bestehen aus vielen parallel arbeitenden einheiten, wobei jede eine vektor-zu-skalar funktion repräsentiert. (Neuron nimmt viele werte von vorherigen Einheiten als Input und berechnet eigene Aktivierung)
- Im allgemeinen werden die hidden units als Funktion beschreiben welche auf einen Vektor aus Eingangsdaten  $\mathbf{x}$  eine affine Transformation  $z = W^T x + b$  durchführt. Dabei beschreibt  $W$  eine Gewichtsmatrix und  $b$  einen Bias-Vektor. Anschließend wird eine elementweise nichtlineare Transformation  $g(z)$  angewendet. Die Transformation  $g(z)$  wird aktivierungsfunktion genannt.
- hidden unit für jede Ebene werden somit durch  $h^{(l)} = g(W^{(l)T} x + b^{(l)})$  beschrieben, wobei  $l$  die jeweilige Ebene vorgibt.
- häufig werden rectified linear units verwendet mit der aktivierungsfunktion  $g(z) = \max\{0, z\}$ .
- Erweiterung ist die leaky ReLU  $f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{otherwise} \end{cases}$
- Es existieren viele weitere Aktivierungsfunktionen.

In diesem Kapitel werden wichtigsten Grundlagen von neuronalen Netzen (NN) erläutert. Das Ziel von neuronalen Netzen ist es eine Funktion  $f^*$  zu approximieren. Für eine Klassifikation beispielsweise mappt die Funktion  $y = f^*(x)$  die Eingangsdaten  $x$  auf eine Klasse  $y$ . Das Mapping in einem neuronalen Netz ist definiert als  $y = f(x; \theta)$ , wobei das Netz die Werte der Parameter  $\theta$  lernt um die Funktion bestmöglichst zu approximieren. Die Zusammensetzung der Funktionen in einem NN können als Model eines gerichteten Graphen betrachtet werden. Ein Beispiel ist die Verkettung von  $f^{(1)}$ ,  $f^{(2)}$  und  $f^{(3)}$  zu  $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$ . Die Funktionen  $f^{(i)}$  beschreiben dabei die verschiedenen Ebenen in einem NN. Während des Trainings wird versucht  $f(x)$  so zu berechnen das sie zu  $f^*(x)$  passt. Das Trainingssample aus  $(x, y)$  gibt dabei vor, dass

die Ausgabeebene des Netzes einen Wert produzieren soll, welcher möglichst ähnlich zu  $y$  ist. Das Verhalten der inneren Ebenen wird nicht direkt vorgegeben. Der Trainingsalgorithmus muss dann entscheiden wie die inneren Ebenen zu verwenden sind um  $f^*$  zu approximieren und die gewünschte Ausgabe zu erhalten.

Hierzu wird beispielhaft ein simples feedforward NN betrachtet. Der Aufbau wird in Abbildung XYZ veranschaulicht. Ein CNN besteht aus einer **Eingangsebene**, mehreren **versteckten Ebenen** im inneren und einer **Ausgangsebene**.

- input
- hidden layer
- output softmax
- convolution discrete equaltion, maybe image, kernel is learning filter, imgage of real-world filter
- activation sigmoid, relu
- striding
- flat
- fully connected layer
- Training (SGD)

## 2.3 Semi-überwachtes Lernen

Das Themengebiet maschinelles Lernen umfasst zwei grundsätzlich verschiedene Aufgabentypen, **unüberwachtes** und **überwachtes** Lernen. Für unüberwachtes Lernen wird ein Set  $X = (x_1, \dots, x_n)$  aus  $n$  Exemplaren/Datenpunkten/Samples benötigt, wobei  $x_i \in \mathcal{X}$  für alle  $i \in [n] := 1, \dots, n$ . Es wird davon ausgegangen, dass die Punkte unabhängig und gleichverteilt (i.i.d.) von einer Verteilung  $\mathcal{X}$  mit der Dichte  $p(x)$  gezogen werden. Jeder Eingang  $x_i$  stellt ein separates Feature darstellt. In einem Bild beispielsweise sind die Features die jeweiligen Werte der Pixel. Das Ziel von unüberwachtem Lernen ist es interessante Strukturen in den Daten  $X$  zu finden. Dabei ist es üblich die Dichte oder ein bekanntes Funktional zu schätzen welches mit hoher Wahrscheinlichkeit  $X$  generiert. Viele Techniken für die Dichteschätzung benötigen eine Latentvariable (unüberwachtes Klassenlabel)  $y$ . Die Latentvariable  $y$  ist eine für das Problem entsprechend modellierte Größe und nicht zu vergleichen mit Klassenlabel bei einer Klassifikation, welches real Klasse widerspiegeln soll.  $y$  wird anschließend verwendet um  $P(x)$  als eine gemischte Verteilung  $\sum_{y=1}^M P(x|y)P(y)$  zu schätzen. Weitere Anwendungen für unüberwachtes Lernen sind beispielsweise Clustering, Outliererkennung oder Dimensionsreduzierung.

Im Gegensatz zu unüberwachtem Lernen benötigt überwachtes Lernen ein Trainingsdatensatz aus Paaren von  $(x_i, y_i)$ . Dabei beschreibt  $y_i$  das Klassenlabel oder das Ziel von  $x_i$ . Das Ziel von überwachtem Lernen ist die Schätzung eines funktionalen Zusammenhangs bzw. ein Mapping von  $x \rightarrow y$ . Dabei gilt es die Wahrscheinlichkeit des Klassifikationsfehlers zu minimieren. Bei einer Klassifikation kann so durch Schätzung der Dichte  $p(y|x)$  den Eingangsdaten  $X$  die zugehörigen Klassenlabels  $y$  zugewiesen werden. In dieser Arbeit wird das überwachte Lernen verwendet um **Convolutional Neural Networks** zu trainieren. Ziel des Netzes ist es eine Funktion  $f^*$  zu approximieren. Für eine Klassifikation mappt die Funktion  $y = f^*(x)$  die Eingangsdaten  $x$  auf eine Klasse  $y$ . Das Mapping in einem neuronalen Netz ist definiert als  $y = f(x; \theta)$  wobei das Netz die Werte der Parameter  $\theta$  lernt um die Funktion bestmöglichst zu approximieren.

Ein weiterer Aufgabentyp ist das **semi-überwachte Lernen** (SSL), welcher zur Kategorie des überwachten Lernens gehört, da auch hier das Ziel ist den Klassifikationsfehler zu minimieren. Wie zuvor wird ein Datensatz mit den zugehörigen Labels  $D_l = \{(x_i, y_i) | i = 1, \dots, n\}$  benötigt, welche i.i.d. von  $P(x, y)$  gezogen wurden. Darüber hinaus existiert ein Datensatz ohne Klassenlabel  $D_u = \{x_{n+j} | j = 1, \dots, m\}$  aus der Verteilung  $P(x)$ . Semi-überwachtes Lernen ist besonders interessant wenn  $m \gg n$ , da ungelabelte Daten oftmals günstiger und einfacher zu erhalten sind als gelabelte Daten. Das Vorgehen bei SSL besteht aus zwei Schritten. Im ersten werden Methoden des unüberwachten Lernens verwendet um mit Hilfe einer Latentvariabel  $y$  die Verteilung  $P(x)$  zu schätzen. Anschließend können die Latentgruppen mit den beobachteten Klassen aus  $D_l$  verbunden werden. Dieses Vorgehen wird bei den Verfahren **Variational Autoencoder** und **Generative Adversarial Network** verwendet.

[2]

### 2.3.1 Generative Netze

Das Ziel von maschinellem Lernen ist die Approximation eines Modells das möglichst allgemein gilt und somit neue bisher unbekannte Daten verarbeiten kann. Der beste Weg solch ein Modell zu verallgemeinern ist es, dass Modell mit mehr Daten zu trainieren. Allerdings ist die Menge der Trainingsdaten oft limitiert. Aus diesem Grund können Fake-Daten generiert werden um den Datensatz künstlich zu erweitern. Bei einer Klassifikation beispielsweise soll den Eingangsdaten  $\mathbf{x}$  ein Label  $y$  zugewiesen werden. Das bedeutet der Klassifikator muss möglichst invariant gegenüber einer breiten Reihe von Transformationen sein. Neue Trainingsdaten  $(\mathbf{x}, y)$  können somit einfach durch eine Transformation der Eingangsdaten  $\mathbf{x}$  aus dem ursprünglichen Trainingsset erzeugt werden. Bekannte Methoden sind z.B. Umwandlung einiger Pixel in einem Bild, Rotation des Bildes oder eine Veränderung der Skalierung des Bildes. Auch kann es bereits genügen den Eingangsdaten ein Rauschen hinzuzufügen um den Datensatz zu erweitern.

[2]

Darüber hinaus existieren weitere komplexere Methoden um Fake-Daten künstlich zu erzeugen. Diese werden generative Modelle genannt. Ziel ist es neue, ungesehene Da-

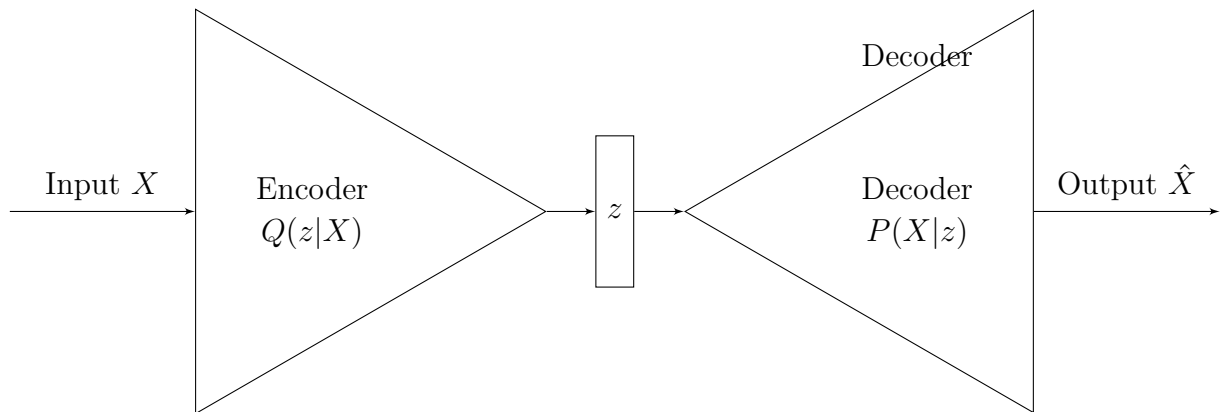


Bild 2.1: Blockschaltbild eines Autoencoders

ten zu erzeugen die ähnlich zu den ursprünglichen Daten sind aber nicht genau gleich sind. Dazu wird ein Set aus Daten  $X$  mit einer unbekannten Verteilung  $P_{unb.}(X)$  benötigt. Es gilt ein Model zu trainieren welches eine Verteilung  $P(X)$  lernen soll, sodass  $P(X)$  möglichst ähnlich zu  $P_{unb.}(X)$  ist. Aus dieser gelernten Verteilung  $P(X)$  kann gesampelt werden um neue Daten zu erzeugen. In dieser Arbeit werden zwei Arten von generativen Netzen untersucht, der "variational autoencoder" und das "generative adversarial network".

[1]

### Variational Autoencoder

Die Grundidee eines Autoencoders (AE) ist die Eingangsdaten über einen Zwischenraum auf die Ausgangsdaten zu kopieren. Dabei beschreibt der Zwischenraum einen Code welcher die Eingangsdaten repräsentiert. Der AE besteht aus zwei hintereinandergeschalteten Netzen. Der strukturelle Aufbau wird in Abbildung 2.1 dargestellt. Das erste Netz, der **Encoder**  $Q$  ist eine Funktion  $z = f(X)$  welche die Eingangsdaten  $X$  in einen Zwischenraum  $z$  mappt. Der Zwischenraum wird im folgenden als Latentraum bezeichnet. Das zweite Netz, der **Decoder**  $P$  soll aus den kodierten Daten  $z$  die ursprünglichen Daten  $X$  rekonstruieren. Der Autoencoder soll dabei nicht nur das Mapping  $g(f(x)) = x$  lernen. Oft wird der AE so eingeschränkt das er wichtige Eigenschaften der Eingangsdaten lernt. So kann der AE beispielsweise die Verteilung  $P(X)$  der Eingangsdaten lernen.

Der **Latentraum** ist eine Vektor  $\mathbf{z}$  aus sogenannten Latentvariablen  $z$ . Dabei können Samples von  $\mathbf{z}$  entsprechend einer Wahrscheinlichkeitsdichtefunktion (PDF)  $P(\mathbf{z})$  entnommen werden. Im weiteren Verlauf wird  $\mathbf{z}$  als normal verteilt angenommen, d.h.  $\mathbf{z} \sim \mathcal{N}(0, I)$ , wobei  $I$  die Einheitsmatrix darstellt. Die multidimensionale Verteilung von  $P(\mathbf{z})$  in  $d$  Dimensionen setzt sich somit aus einem Set von  $d$  normal verteilten Variablen  $z$  zusammen.

Wie bereits erwähnt ist das Ziel ein Model der Daten zu finden, d.h. der AE soll die Verteilung  $P(X)$  der Eingangsdaten  $X$  lernen. Dazu wird eine Funktion  $X = f(z; \theta)$  benötigt, welche die Latentvariablen  $z$  nach  $X$  mappt. Der Vektor  $\theta$  definiert die Pa-

parameter der Netzes. Es gilt  $\theta$  so zu optimieren, dass durch ziehen von Samples  $z$  aus  $P(z)$  die Funktion  $f(z; \theta)$  mit hoher Wahrscheinlichkeit Ausgaben produziert welche gleich den  $X$ 's aus dem Datenset sind. Dieser Zusammenhang kann nach dem Gesetz der totalen Wahrscheinlichkeit beschreiben werden als:

$$P(X) = \int \underbrace{P(X/z, \theta)}_{f(z; \theta)} P(z) dz. \quad (2.6)$$

Wenn das Model mit hoher Wahrscheinlichkeit Beispiele aus dem Trainingsdatensatz produziert ist es ebenfalls sehr wahrscheinlich, dass das Model Samples erzeugen kann die ähnlich aber nicht gleich den Trainingsdaten sind.

Das bisherige Konzept wird im Folgenden zu einem variational Autocoder (VAE) erweitert. Hierbei ist das Ziel die Verteilung  $P(z)$  aus  $P(z|X)$  zu erschließen. Dies macht Sinn, da  $z$  so eine Verteilung annimmt welche mit hoher Wahrscheinlichkeit  $X$  produzieren kann. Diese Methode wird variational inference (VI) genannt. Die Idee ist die tatsächliche Verteilung  $P(z|X)$  mittels einer einfacheren Verteilung  $Q(z|X)$  zu modellieren, z.B. eine Gaussverteilung. Dabei gilt es den Unterschied zwischen der tatsächlichen und der modellierten Verteilung zu minimieren. Dazu wird die Kullback-Leibler (KL) Divergenz verwendet. Diese Metric beschreibt den Unterschied zweier Verteilungen, hier den Unterschied zwischen  $P(z|X)$  und  $Q(z|X)$ . Die KL Divergenz wird formliert als:

$$D_{KL}[Q(z|X)||P(z|X)] = E[\log Q(z|X) - \log P(z|X)]. \quad (2.7)$$

Mit der Anwendung der Bayes-Regel und einigen Transformationen ergibt sich daraus die Kostenfunktion für den VAE als:

$$\log P(X) - D_{KL}[Q(z|X)||P(z|X)] = E[\log P(X|z)] - D_{KL}[Q(z|X)||P(z)]. \quad (2.8)$$

Die linke Seite der Gleichung 2.8 beschreibt die kostenfunktion des VAE. Wie bereits erwähnt ist das oberste Ziel des VAE die Wahrscheinlichkeit von  $P(X)$  zu maximieren. Hier beschreiben durch die log-Wahrscheinlichkeit  $\log P(X)$ . Hinzu kommt der Regularisierungsterm  $D_{KL}[Q(z|X)||P(z|X)]$  welcher sicherstellt, dass  $Q$  Latentvariablen  $z$  produzieren kann aus denen sich anschließend  $X$  reproduzieren lässt. Die rechte Seite von Gleichung 2.8 ist eine Darstellung die mittels stochastic gradient descent Methoden optimiert werden kann. Wie im oberen Teil bereits bschreiben wird die Verteilung von  $P(z)$  als Standartnormalverteilung  $\mathcal{N}(0, I)$  definiert. Somit soll auch  $Q(z|X)$  eine Gaussverteilung mit Erwartungswert  $\mu(X)$  und Varianz  $\Sigma(X)$  werden. Für die KL Divergenz ergibt sich durch Einsetzen diesen beiden Verteilungen:

$$D_{KL}[N(\mu(X), \Sigma(X))||N(0, I)] = \frac{1}{2} \sum_k \left( \Sigma(X) + \mu^2(X) - 1 - \log \Sigma(X) \right). \quad (2.9)$$

Jetzt folgt noch der Reparametrisierungs Trick!!!!!!

**GAN**

TODO

### **2.3.2 Attack-Defence**

## **2.4 Akustic-Preprocessing**



## **3 Ergebnisse**

TODO

### **3.1 Autoencoder**

TODO

### **3.2 GAN**

TODO

### **3.3 Attak-Defence**

TODO

## 4 Zusammenfassung

TODO

# Literaturverzeichnis

- [1] Doersch, C.: Tutorial on Variational Autoencoders, *ArXiv e-prints*, Juni 2016
- [2] Goodfellow, I.; Bengio, Y.; Courville, A.: *Deep Learning*, MIT Press, <http://www.deeplearningbook.org>, 2016

# A Inhaltliche Ergänzungen

Im Anhang findet ergänzender Inhalt Platz. Hierzu zählen beispielsweise Quellcode, Tabellen und Abbildungen.

Ein Quellcode kann in Auszügen mittels der “listing”-Umgebung eingefügt werden.

```
1 use Math::Derivative qw(Derivative1 Derivative2);
2 @dydx=Derivative1(\@x,\@y);
3 @d2ydx2=Derivative2(\@x,\@y);
4 @d2ydx2=Derivative2(\@x,\@y,$yp0,$ypn);
```

**Quellcode A.1:** Beispielbeschriftung

Eine Referenz auf Quellcodeauszug A.1 muss immer vorhanden sein.