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

Lateinische Buchstaben

g	m/s^2	Erdbeschleunigung
H	m	(Fall)Höhe

Griechische Buchstaben

Δ	%	Abweichung
ϵ	m^2/s^3	isotrope Dissipationsrate

Tiefgestellte Indizes

1,2	Eintritt, Austritt
abs	absolut
hyd	hydraulisch

Abkürzungen

BP	Betriebspunkt
DT	(Teil)Rechengebiet Saugrohr

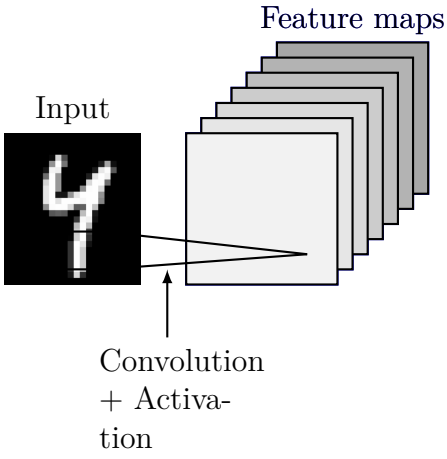
1 Einleitung

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, wobie jede eine vektor-zu-skalar funktion repräsentiert. (Neuron nimmt viele werte von vorherigen EInheiten als Input un 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. Fomal wird die Kostenfunktion beschreiben als:

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

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

Output units:
- Output units eng gekopppelt 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 vervollständign, ensprechend 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 ouputunits werden oft verwendet um Erwartungswert einer bedingten Gaussverteilung zu schätzen:

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

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

- (3.10) logistic sigmoid:

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

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

—

- 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 Verallgemeienrung der sigmoid funktion. Meistens wird die softmax Funktion als Ausgang einer Klassifikation verwendet um die Wahrscheinlichkeitsverteilung über n verschiedenen Klassen zu repäsentieren.
- generalizierung der sigmoid funktion zur schätzung der bernoulliverteilung. ergibt $\hat{y}_i = P(y = i|x)$. Dabei soll nciht nur jedes Element von $\hat{y}_i$ zwischen 0 und 1 liegen sonder 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: $\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_j e^{z_i}}$

- vllt beispiel aus Präsi wie beliebige Werte über die softmax in Verteilung umgerechnt werden + Bild von MNIST mit zugehörigem Blakendiagram 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 jeweile 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 $f(x) = \begin{cases} x & if x > 0 \\ \alpha x & otherwise \end{cases}$
- Darüberhinaus existieren viele weitere Aktivierungsfunktionen.

Conv netzworks:
- Erweiterung der bisher beschreibenen 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 convolituional 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:

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

- Bei der Anwendung von CNNs ist der Eingang typischerweise ein multidimensionaler Array aus Daten. Der Kernel beschreibt ein mutlidimensionalen Array aus Parameter die während des Trainings vom Lernalgorithmus angepaqsst werden.
- oftmals wird convolution über mehr als eine Achse berechnet. Für ein zweidimensionals Bild I als Input beispielsweise kann ein zweidimensionaler Kernel K verwendet werden. Die Convolution resultiert in:

$$S(i,j) = (I \star K)(i,j) = \sum_{m} \sum_{n} I(m,n)K(i-m, j-n). \tag{2.5}$$

- Erwähnen, dass für die Implementierung einige Umformungen angewendet werden, flipping; cross-correlation - Bild von Faltungsprozess und Referenze einfügen.
Motivation
- CNNs haben drei Ideen welche heflen 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 Paramater. 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 gemsaten 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 traditionlellen 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. Somit wird an Stelle eines seperaten Parameterset für jede Position im Input nur ein Parameterset für alle Positionen benötigt.
- 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.
- Beispielsweise soll die Funktion I die Pixel eines Bildes beschreiben. Das Mapping auf eine andere Funktion soll beschreiben werden durch $I' = g(I)$, wobei gilt $I'(x, y) = I(x - 1, y)$. Dadurch wird jedes Pixel in dem Orginalbild um eine Einheit verschoben. Durch die equivariance Eingenschaft ergibt sich der selbe Output wenn zuerst die Transformation g und dann die Convolution angewendet wird bzw. erst die Conv, dann die Trafo g.
- Diese Eigenschaft ist beispielsweise in der Bildverarbeitung sehr hilfreich. Hier kann die erste Conv-Ebene Feautres wie z.B. Ecken oder Kanten erkennen. Da Ecken und Kanten über das gesamte Bild verteilt auftreten können ist es hilfreich die Paramter über das gesamte Bild zu teilen.

Pooling -Der Conv-Oprator tritt oftmals in Verbindung mit dem Pooling-Operator auf. Das Pooling wird auf den Output der Conv-Ebene angewendet und ersetzt diesen durch eine Zusammnfassung der nahegelegenen Outputs. Der max Pooling-Operator beispielsweise fasst eine rechteckige Umgebung der Outputs zusammen durch Ausgabe des maximalen Werts in dieser Umgebung.
- Darüberhinaus existiren viele weitere Pooling-Verfahren.
- Ziel des Pooling ist es die Filter invariant gegnüber kleinen Veränderungen im Input zu machen. Invariant bedeutet, dass durch eine kleine Änderung des Inputs die meisten pooled Outputs unverändert bleiben.
- Diese Eingenschaft ist nützlich, da es oftmals von größerem Interesse ist zu erkennen ob Features existieren anstatt deren exakte Position zu erhalten.
- Beispielsweise in der Gesichtserkennung ist es ausreichen zu erkennen, dass sich ein Auge auf der rechten Seite und eines auf der linken Seite befindet, anstatt die Position der Augen auf das pixel genau zu bestimmen. - Auch ist es möglich die Pooling-Region um jeweils k Pixel zu verschieben. Dies führt dazu, dass die Anzahl der Output units ca. k-mal weniger werden und somit deutlich weniger Inputs in der nächsten Ebene verarbeitet werden müssen. Dadurch kann der Rechenaufwand des Netzes verringert werden.
goodfellow S.350

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, wobie jede eine vektor-zu-skalar funktion repräsentiert. (Neuron nimmt viele werte von vorherigen EInheiten als Input un 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 jeweile Ebene vorgibt.

- häufig werden rectified linear units verwendet mit der aktivierungsfunktion $g(z) = max\{0, z\}$.
- Erweiterung ist die leaky ReLU $f(x) = \begin{cases} x & if x > 0 \\ \alpha x & otherwise \end{cases}$
- 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 θ 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, **unübrwachtes** und **übrwachtes** Lernen. Für unüberwachtes Lernen wird ein Set $X = (x_1, \ldots, x_n)$ aus n Exmplaren/Datenpunkten/Samples benötigt, wobei $x_i \in \mathcal{X}$ für alle $i \in [n] := 1, \ldots, n$. 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 seperates Feature darstellt. In einem Bild beispielsweise sind die Features die jeweiligen Werte der Pixel. Das Zeil 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 wiederspiegeln 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 biespielsweise Clustering, Outliererkennung oder Dimensionsreduzierung.
Im Gegensatz zu unüberwachtem Lernen benötigt überwachtes Lernen ein Trainingsdatenset 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 $x \to y$. 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 θ lernt um die Funktion bestmöglichst zu approximieren.
Ein weiterer Aufgabentyp ist das **semi-überwachte** Lernen (SSL), welcher zur kathegorie des des überwachten Lernens gehört, da auch hier das Ziel ist den Klassifikationsfehler zu minimieren. Wie zuvor wird ein Datenset mit den zugehörigen Labeln $D_l = \{(x_i, y_i)|i = 1, \ldots, n\}$ benötigt, welche i.i.d. von $P(x, y)$ gezogn wurden. Darüber hinaus existiert ein Datenset ohne Klassenlabel $D_u = \{x_{n+j}|j = 1, \ldots, m\}$ aus der

Verteilung$P(x)$. Semi-überwachtes Lernen ist besonders interessant wenn $m >> n$, da ungelabelete Daten oftmals günstiger und einfacher zu erhalten sind als gelabelte Daten. Das vorgehen bei SSL bsteht 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 Model zu verallgemeinern ist es, dass Model 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 $\mathbf{x}$ eine 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 Daten 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 gesamplet werden um neue Daten zu erzeugen. In dieser Arbeit werden zwei Arten von generativen Netzen untersucht, der "variational autoencoderünd 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 hintereiander gschalteten Netzen. Der strukturelle Aubau wird in Abbildung 2.1 dargestellt. Das erste Netz, der **Encoder** Q ist eine Funktion $z = f(X)$ welche die Eingangsdaten

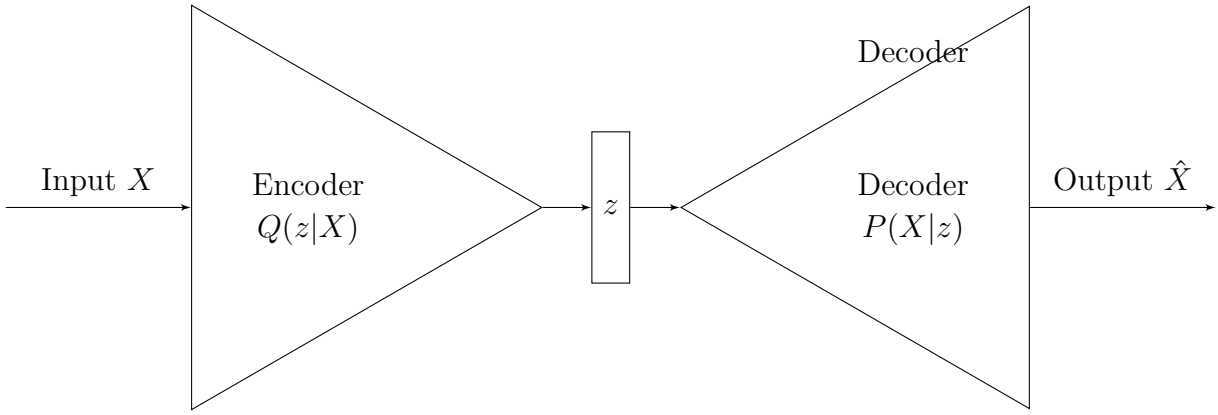


Bild 2.1: Blockschaltbild eines Autoencoders

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 beispilsweise 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 breits 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 θ definiert die Parameter der Netzes. Es gilt θ 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:

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

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 Autoncoder (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:

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

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

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

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 Regularizierungsterm $D_{KL}\Big[Q(z|X)||P(z|X)\Big]$ 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:

$$D_{KL}\Big[N(\mu(X), \Sigma(X))||N(0, I)\Big] = \frac{1}{2}\sum_{k}\Big(\Sigma(X) + \mu^2(X) - 1 - \log\Sigma(X)\Big). \tag{2.9}$$

- Um den Eingang z für den Encoder zu erhalten müssen Sample aus eine Gaussverteilung gezogen werden, welche durch die Parameter aus dem DEcoder definiert wird. Dies führt allerdings zu Problemen bei der Optimierung mit gradient descent, da der Sampling-Prozess keine Gradienten besitzt.
- Damit das Netz differenzierbar bleib wird der sogenannte Reparametrisierungstrick angewandt. - Eine Standartnormalverteilung lässt sich in jede Gaussverteilung konvertieren, falls deren Erwartungswert und Varianz bekannt sind. Somit genügt es Sample aus einer Standartnormalverteilung $\epsilon Tilde N(0,1)$ zu ziehen und diese mit dem Erwartungswert μ und der Varianz Σ aus dem Encoder nach z zu konvertieren:

$$z = \mu(X) + \Sigma^{\frac{1}{2}}(X)\epsilon. \tag{2.10}$$

Somit wird der eigentliche Sampling-Prozess aus dem Netz herausgezogen. Lediglich μ und Σ sind noch Teil des Netzes und die Backpropagation kann problemlos durchgeführt werden.

GAN

TODO

2.3.2 Attack-Defence

2.4 Akustic-Preprocessing

Dieses Kapitel beschreibt die

3 Ergebnisse

TODO

3.1 Baseline

TODO

3.2 Autoencoder

TODO

3.3 GAN

TODO

3.4 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.