Informatik
Hauptcampus

H O C H
S C H U L E
T R I E R

TensorFlow: Prinzip und Anwendung von Autoencodern

Denis Wagner

Seminar “Maschinelle Lernverfahren“

Betreuer: Herr Prof. Lohscheller

Trier, 22.01.2018

Kurzfassung

Diese Arbeit befasst sich mit neuronalen Netzwerken, oder präziser, Autoencodern. Es wird der Aufbau und das Training eines generellen Autoencoder definiert und auch weitere Varianten, die das Grundprinzip dieser neuronalen Netze erweitern und für eine weitreichende Anzahl von Anwendungsgebieten anpassen. Wichtige Merkmale verschiedener Varianten werden an einigen Stellen durch Beispielimplementierungen mit Hilfe der Python-Bibliothek TensorFlow programmiertechnisch veranschaulicht. Des Weiteren werden auch konkrete Anwendungsbeispiele genannt, in denen die verschiedenen Varianten von Autoencodern in einem praktischen Kontext dargestellt werden.

Inhaltsverzeichnis

1

Einleitung

Das Lernen gehört zweifellos zu den wichtigsten Eigenschaften einer intelligenten Lebensform. Es dient dem Aneignen von Fähigkeiten und dem Anpassen an Gegebenheiten, die nicht, oder nicht gänzlich, vorhersehbar sind. Dies gilt natürlich auch für künstliche Intelligenzen, die anstatt einer biologischen Grundlage eine elektronische besitzen. Sie müssen sich, wie auch der Mensch selbst, gewissen Herausforderungen stellen können, ohne dass jedes einzelne Szenario geplant worden ist. Denn ein System, welches für jedes nur erdenkliche Problem eine Lösung kennt, ist - zumindest aus heutiger Sicht - undenkbar. Allerdings ist eine der bedeutendsten Fähigkeiten des Menschen, ein Verhalten zu produzieren, welches vorher noch unbekannt war: die Kreativität. Da eine Maschine ein meist deterministisches Verhalten besitzt, ist es schwer, sich dies bei einem Computer vorzustellen. Doch neuronale Netze sind der erste Schritt genau das zu erreichen. Sie sind eine sehr menschenähnliche Darstellung eines Gehirns für Maschinen, was sich Autoencoder zu Nutze machen, um diesem Ziel näher zu kommen. Denn das ist es, was der Vater der Künstlichen Intelligenz, John McCarthy, mit seiner Idee erreichen wollte (nach [PN12]).

Diese Arbeit beschreibt detailliert die Funktionen und weitreichenden Anwendungsgebiete von Autoencodern. Dies beinhaltet zum einen, dass der *Autoencoder* ein Überbegriff für eine Reihe von neuronalen Netzen ist, die das gleiche Grundprinzip verfolgen, aber auch, dass eine bestimmte Variation eines Autoencoders in verschiedenen Bereichen unterschiedlich angewendet werden kann.

1.1 Grundprinzip von Autoencodern

Der Autoencoder nimmt eine Eingabe x aus einer bestimmten Anzahl von Datenpunkten, komprimiert diese zu einer latenten Darstellung z und versucht diese wieder auf die ursprüngliche Anzahl von Datenpunkten zu rekonstruieren und erzeugt somit die Ausgabe x', sodass $x \approx x'$ gilt [IG16]. Wie in Abbildung 1.1 zu sehen ist, muss x' aus z abgeleitet werden. Wenn z genauso viele Datenpunkte besitzt wie

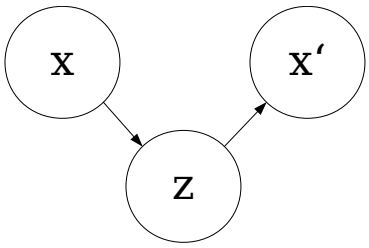


Abb. 1.1. Prinzip der Funktionsweise eines Autoencoders

x, würde $x = x'$ gelten und der Autoencoder nach [Gal16] die Identitätsfunktion erlernen. Viel interessanter ist allerdings der Fall, wenn z aus deutlich weniger Datenpunkten besteht als x. Durch die fehlenden Datenpunkte in z treten bei der Rekonstruktion von x' Abweichungen auf, sodass $x = x'$ nicht mehr gilt. Allerdings versucht der Autoencoder sich dem x anzunähern, sodass am Ende $x \approx x'$ gilt. Diese Approximation zwischen x und x' hat zur Folge, dass eine ähnliche Ausgabe zu einer Eingabe generiert wird. Je nach Training kann diese Approximation nach [IG16] in eine ganz bestimmte Richtung gelenkt werden, sodass nur gewisse Bereiche Abweichungen aufweisen oder die Abweichung nur in einer gewissen Art erfolgt. Der Autoencoder verändert die Eingabedaten also nicht zufällig, sondern einem bestimmten Muster entsprechend. Ein Autoencoder der zum Beispiel mit Bildern von Gesichtern trainiert wird, kann diese Gesichter abändern, ohne die Grundstruktur eines Gesichtes zu zerstören. Somit ist dieser Autoencoder in der Lage, "kreativ" zufällige Gesichter zu generieren. Dieses Prinzip kann auf diverse Eingaben übertragen werden. Eine weitere Anwendung dieses Prinzips nach [Gal16] ist, dass der Rekonstruktionsschritt nach dem Training eliminiert und die latente Darstellung z als Ausgabe verwendet wird, um zum Beispiel eine Dimensionsreduktion (Verkleinerung von Bildern, etc.) zu erreichen.

2

Grundlagen eines Autoencoders

In diesem Kapitel wird auf die grundlegende Funktionsweise eines Autoencoders (im Folgenden *AE* genannt) eingegangen sowie die einführende Idee des AE konkretisiert. Außerdem werden Trainingsmethoden aufgezeigt, die es dem AE ermöglicht, seine Fähigkeiten zu erlangen.

2.1 Funktionsweise und Aufbau

Ein AE ist nach [Gal16] ein neuronales Netzwerk, welches dazu trainiert wird, seine Eingabe zu reproduzieren. Die Funktion eines AE besteht, wie der Name bereits vermuten lässt, darin, dass er eine gegebene Eingabe automatisch *encodieren* und auch wieder *decodieren* kann. Folglich muss es einen *Encoder* und einen *Decoder* geben. Dies sind zunächst zwei separate einschichtige Netze, welche allerdings zusammengeführt werden, indem der Output des Encoders mit dem Input des Decoders verbunden wird. Dies resultiert in einem Netz, welches sich mit drei Schichten repräsentieren lässt:

- Eingabeschicht (engl. *input layer*)
- Verdeckte Schicht (engl. *hidden layer*)
- Ausgabeschicht (engl. *output layer*)

Die wichtigste Eigenschaft eines Autoencoders, liegt in der Anzahl von Neuronen in der Eingabe- und Ausgabeschicht. Diese müssen exakt gleich sein, da die Ausgabe eine Rekonstruktion der Eingabe repräsentieren soll. Die verdeckte Schicht z, aus Abbildung 1.1, stellt den Encoder, die Ausgabeschicht x' den Decoder dar. Die verdeckte Schicht z beinhaltet die *latente Repräsentation* der Eingabedaten in komprimierter Form. Sie ist die wichtigste Komponente des AE, der nach [Gal16] für das eigentliche Encodieren der Eingabedaten verantwortlich ist. Die Eingabeschicht hingegen ist nur eine Schnittstelle, damit Eingabedaten zum Encoder gelangen. Somit kann das neuronale Netz, welches den AE repräsentiert, nach [Caw03], auch nur als zweischichtiges Netz betrachtet werden, da die Eingabeschicht keine besondere Funktion besitzt (im Folgenden wird ein allgemeiner AE als zweischichtiges Netz bezeichnet). Das, was einen AE nun von einem normalen mehrschichtigen Perzeptron (engl. *multilayer perceptron*, im Folgenden *MLP*

genannt) unterscheidet, ist die Tatsache, dass die Ausgabeschicht die gleiche Anzahl von Neuronen besitzt wie die Eingabeschicht. Nach [Gal16] gehört ein AE zur Familie der *unüberwachten Lernalgorithmen* (engl. *unsupervised learning*), da er selbst entscheidet, ob eine erzeugte Ausgabe nun gut ist oder nicht. Im Gegensatz dazu gehört ein MLP nach [PN12] generell zu den überwachten Lernalgorithmen (engl. *supervised learning*), da es beschriftete Daten (engl. *labeled data*) verwendet, um dessen Ausgaben zu überprüfen.

Neuronale Netze werden nach [Cri17] mithilfe des linearen Modells (engl. *linear model*) mathematisch beschrieben. Da ein AE auch ein neuronales Netz ist, kann nach [Gal16] auch dieser wie folgt definiert werden:

Definition 2.1. *Autoencoder*
$\forall\ d, O_{di}, I_{di}, i \in \mathbb{N}$:

- $x \in [0,1]^d$ *ist der Eingabevektor (engl. input vector), der beschreibt, ob ein Eingang aktiv (1) oder inaktiv (0) ist. d beschreibt die Dimension, also die Anzahl von Neuronen, die diese Schicht besitzt.*

- $W_i \in \mathbb{R}^{I_{di} \times O_{di}}$ *ist eine Matrix, die die Gewichtungen des Inputs projiziert auf den Output der i-ten Schicht darstellt.* I_{di} *ist dabei die Dimension des Eingabevektors und* O_{di} *die des Ausgabevektors (engl. output vector) der i-ten Schicht.*

- $b_i \in \mathbb{R}^{O_{di}}$ *ist der Bias-Vektor. Dieser wird benutzt, um das Verhalten des neuronalen Netzes um einen konstanten Wert zu verändern. Allerdings kann dieser auch dynamisch durch das Training angepasst werden, ist aber danach nicht mehr von den Eingaben abhängig und fungiert als Konstante.*

- $a(h)$ *ist die Aktivierungsfunktion, die entscheidet, ob ein Neuron durch einen gegebenen Input aktiviert wird. h stellt die Eingabewerte für ein Neuron dar.*

Allgemein kann daher, nach [Gal16], ein AE beschreiben werden mit:

$$z = a(x \cdot W_1 + b_1)$$
$$x' = a(z \cdot W_2 + b_2)$$

Die Anzahl der Gewichtungsmatrizen hängt mit der Anzahl der Schichten zusammen. Wenn es also, wie bei einem allgemeinen AE, zwei Schichten gibt, existieren auch zwei Gewichtungsmatrizen. Einmal die Projektion W_1 von dem Input Layer x auf den Hidden Layer z und W_2 dem Hidden Layer z auf den Output Layer x':

$$W_1 = \begin{pmatrix} w_{1_{1,1}} & w_{1_{2,1}} & \dots & w_{1_{x_d,1}} \\ w_{1_{1,2}} & w_{1_{2,2}} & \dots & w_{1_{x_d,2}} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1_{1,z_d}} & w_{1_{2,z_d}} & \dots & w_{1_{x_d,z_d}} \end{pmatrix} \quad W_2 = \begin{pmatrix} w_{2_{1,1}} & w_{2_{1,1}} & \dots & w_{2_{z_d,1}} \\ w_{2_{1,2}} & w_{2_{1,2}} & \dots & w_{2_{z_d,2}} \\ \vdots & \vdots & \ddots & \vdots \\ w_{2_{1,x'_d}} & w_{2_{1,x'_d}} & \dots & w_{2_{z_d,x'_d}} \end{pmatrix}$$

(Matrixdarstellung nach [vA04])

Es gilt also $W_1 \in \mathbb{R}^{x_d \times z_d}$ und $W_2 \in \mathbb{R}^{z_d \times x'_d}$, wobei x_d, z_d und x'_d die Dimensionen der entsprechenden Schichten repräsentieren. Dieses lineare Modell, mit dem der AE beschrieben wurde, kann nun wie in Abbildung 2.1 graphisch veranschaulicht werden.

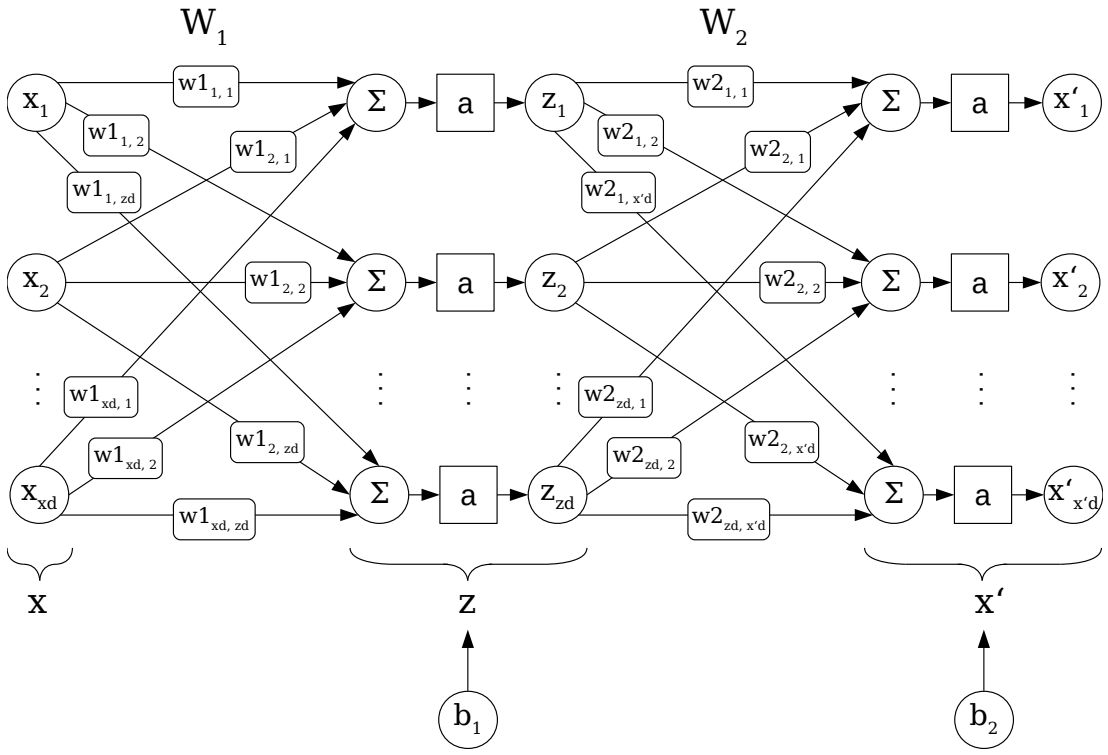


Abb. 2.1. Aufbau eines Autoencoders als neuronales Netzwerk mit den Gewichtungsmatrizen W_1 und W_2

Zu beachten ist in Abbildung 2.1, dass die Werte der Bias-Vektoren b_1 und b_2, ähnlich wie eine zusätzliche Eingabe, zu den entsprechenden Aktivierungsfunktionen hinzugegeben werden, damit die Aktivierung eines einzelnen Neurons konstant beeinflusst werden kann.

Wie am Anfang dieses Kapitels bereits angemerkt, ist die verdeckte Schicht die wichtigste Komponente eines AE, somit haben Veränderungen an dieser große Auswirkungen auf die Funktionalität. Die Dimension der Schicht spielt dabei eine tragende Rolle. So muss bei einem Modell, wie in Abbildung 2.1, z_d nicht unbedingt gleich x_d sein. Wenn die Dimension der verdeckten Schicht kleiner ist als die der Ein- oder Ausgabeschicht, also $z_d < x_d$, dann heißt der AE untervollständig (engl. undercomplete). Und wenn die Dimension der verdeckten Schicht größer ist als die der Ein- oder Ausgabeschicht, also $z_d > x_d$, dann heißt der AE übervollständig (engl. overcomplete) ([IG16]). In der Regel werden untervollständige AEs verwendet, damit diese, nach [IG16], lernen, wichtige Merkmale zu filtern. Bei einem

übervollständigen AE wird, nach [Gal16], ohne weitere Beschränkungen die Identitätsfunktion gelernt, da er die Daten einfach von der Eingabeschicht zur Ausgabeschicht kopieren kann. Wie dies jedoch umgangen wird, folgt im nächsten Kapitel.

2.2 Training

Ein neuronales Netzwerk muss natürlich trainiert werden, damit es "weiß", was es tun soll. Dies erfolgt für gewöhnlich dadurch, dass zu einem Ergebnis ein sogenannter *Verlust* (engl. *loss* oder *error*) bestimmt wird. Bei einem AE wäre dies nach [Gal16] der Rekonstruktionsverlust (engl. *reconstruction error*) zwischen der Eingabe x und der Rekonstruktion x'. Dieser Verlust wird durch eine Verlustfunktion (engl. *loss function*) errechnet. Eine Verlustfunktion kann im Grunde alles sein, allerdings ist das Ziel bei einer solchen Funktion, bei einem bestimmten Loss eine möglichst präzise Änderung der Gewichte des neuronalen Netzes zu erreichen, damit dieser Loss möglichst minimiert wird. Wenn der Unterschied zwischen Eingabe und Ausgabe höher ist, muss eine gute Verlustfunktion auch einen entsprechend höheren Wert zurückgeben, damit das Netz auch für höhere Abweichung eine entsprechende Bestrafung (engl. *panalty*) erhält.

Der Verlust eines AE kann nach [IG16] wie folgt beschrieben werden:

$$L(x, x') \text{ für } x' = decode(encode(x))$$

Dabei ist L die Verlustfunktion, die das Netz bestraft, wenn x' zu unterschiedlich von x ist, zum Beispiel mit Hilfe der mittleren quadratischen Abweichung (engl. *mean squared error*). Die mittlere quadratische Abweichung ist eine nicht-lineare Verlustfunktion, was bedeutet, dass das Netz quadratisch zum eigentlichen Verlust bestraft wird. Also je höher der Verlust, desto stärker steigt die Funktion an, was natürlich hohe Abweichungen deutlich effektiver bestraft als niedrige.

Ein AE extrahiert (wie bei der Hauptkomponentenanalyse (engl. *Principal Component Analysis, PCA*)), nach [Gal16], die latenten beziehungsweise verborgenen Variablen (engl. *latent/hidden variables*) der Eingabedaten. Diese latenten Variablen beschreiben die *Hauptkomponenten* der Eingabe, also bestimmte Komponenten, die eine hohe Signifikanz darstellen. Sie werden latent oder verborgen genannt, da es zunächst nicht immer offensichtlich erscheint, welche Komponenten nun eine hohe Signifikanz besitzen und welche nicht. Oft stellen die latenten Variablen keine konkreten Werte der Eingabedaten dar, sondern bestehen aus einer Kombination aus verschiedenen Werten. Nach ihnen wird dann die latente Repräsentation in der verdeckten Schicht erstellt. Wie in Kapitel 2.1 bereits erwähnt, kann ein untervollständiger AE die wichtigen Merkmale der Eingabedaten filtern. Somit kann, nach [Gal16], zwar nicht die Identitätsfunktion erlernt werden, sondern stattdessen eine komprimierte Repräsentation der Daten. Allerdings kann dies, nach [Gal16], auch mit Hilfe eines übervollständigen AE erreicht werden, indem sogenannte *Sparsity Constraints* hinzugefügt werden, um eine Überanpassung, also

das simple Kopieren der Eingabedaten zur Ausgabeschicht, zu vermeiden ([Gal16]). Durch diese bleiben einige Neuronen der verdeckten Schicht für die meiste Zeit inaktiv, sofern sie für bestimmte Merkmale zur Rekonstruktion nicht benötigt werden. Die Inaktivität des i-ten Neurons hängt von der verwendeten Aktivierungsfunktion $a_i(h)$ ab ([Gal16]). Wenn für die Aktivierungsfunktion $a_i(x) = tanh(x)$ gilt, bedeutet ein Wert nahe -1, nach [Gal16], dass dieses Neuron inaktiv bleibt, wie in Abbildung 2.2 zu erkennen ist.

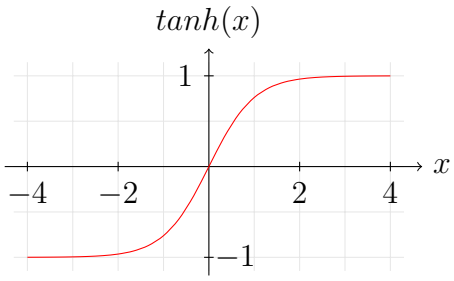


Abb. 2.2. Funktionsverlauf von $tanh(x)$ ([Wei])

Das Verwenden von einem übervollständigen AE mit Sparsity Constraints hat den Vorteil, dass er die gleichen Fähigkeiten besitzt wie ein untervollständiger AE, und noch mehr, da die Neuronen, die bei einem bestimmten Input inaktiv waren, nun für eine andere Art Input verwendet und trainiert werden können, die ein untervollständiger AE gar nicht besitzt. Zu erkennen ist jedoch, dass sowohl die tatsächliche Limitierung der Anzahl von Neuronen als auch das Deaktivieren existierender Neuronen in der verdeckten Schicht zu *Sparsity* (deut. *Mangel*) führt ([Gal16]).

Sparsity kann, nach [Gal16], nun dadurch erreicht werden, dass ein weiterer Term zur Verlustfunktion hinzugerechnet wird: einen Regularisierer (engl. *regularizer*) ([IG16]). Dieser verhindert die Überanpassung und zwingt Neuronen zur Inaktivität. Dafür muss zunächst der durchschnittliche Aktivierungswert $\hat{\rho}_j$ eines Neurons j der verdeckten Schicht (mit (2) markiert) über die gesamte Trainingsmenge (engl. *training set*) TS bestimmt werden:

$$\hat{\rho}_j = \frac{1}{|TS|} \sum_{i=1}^{|TS|} a_j^{(2)}(x_i)$$

Daraus kann, nach [Gal16], nun der *Sparsity Parameter* ρ bestimmt werden, der für alle Neuronen der verdeckten Schicht gilt:

$$\forall j : \hat{\rho}_j \leq \rho$$

Dadurch, dass ρ mindestens genauso groß ist wie ein $\hat{\rho}_j$, kann der Sparsity Parameter dafür genutzt werden, die meisten Neuronen daran zu hindern, dass sie sich aktivieren, sofern dieser auf einen Wert nahe 0 gesetzt wird ([Gal16]).

Als Regularisierer wird, nach [Gal16], in der Regel die *Kullback-Leibler-Divergenz* (im Folgenden KL genannt) verwendet:

$$\sum_{j=1}^{O_{d_2}} \rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_j} = \sum_{j=1}^{O_{d_2}} KL(\rho||\hat{\rho}_j)$$

Diese Divergenz misst die Gleichheit zweier Wahrscheinlichkeitsverteilungen. Vereinfacht bedeutet dies, dass der Unterschied zwischen ρ und $\hat{\rho}_j$ bestimmt wird. Der Unterschied aller Neuronen, die für einen bestimmten Output benötigt werden, ergeben zusammenaddiert die Gesamtabweichung zwischen diesen beiden Verteilungen ([Gal16]). Also gilt auch hier: Je höher der Unterschied eines jeden Neurons ist, desto höher wird auch das Ergebnis der KL und demnach auch die Bestrafung des Netzes ausfallen. Somit muss versucht werden, diesen Regularisierer mit dem Trainingsprozess durch das Gradientenverfahren zu minimieren.

Mit der Verlustfunktion L und dem Regularisierer KL sieht die regularisierte Verlustfunktion LR, nach [Gal16], nun wie folgt aus:

$$LR = L(x, x') + \sum_{j=1}^{O_{d_2}} KL(\rho||\hat{\rho}_j)$$

Die regularisierte Verlustfunktion LR kann sowohl für unter- als auch für übervollständige AEs verwendet werden, da beide Arten der Sparsity (geringere Anzahl von Neuronen in der verdeckten Schicht oder Sparsity Constraints zum Verhindern der Aktivierung von Neuronen) zusammen in einem AE verwendet werden können.

3

Varianten von Autoencodern

Der Begriff *Autoencoder* ist eine Generalisierung sämtlicher künstlicher neuronaler Netze, die dem gleichen Prinzip wie aus Kapitel 2.1 folgen. In den folgenden Unterkapiteln werden einige Variationen eines AE aufgeführt und deren Eigenschaften beschrieben. Außerdem werden grobe Anwendungsbeispiele genannt, um den Nutzen der einzelnen Varianten in einem praktischen Kontext aufzuzeigen. Weitere Anwendungen werden allerdings in Kapitel 4 genauer erläutert.

3.1 Sparsing Autoencoder

Ein *Sparsing Autoencoder* ist ein AE, der sich für seine Funktion die *Sparsity* zu Nutze macht. Dabei ist es unabhängig davon, wie die Sparsity erreicht wird. Eine Variante ist es, die Anzahl der Neuronen in der verdeckten Schicht geringer zu halten als die den anderen Schichten. Dies ist bei einem untervollständigen AE, wie in Abbildung 3.1(a), der Fall.

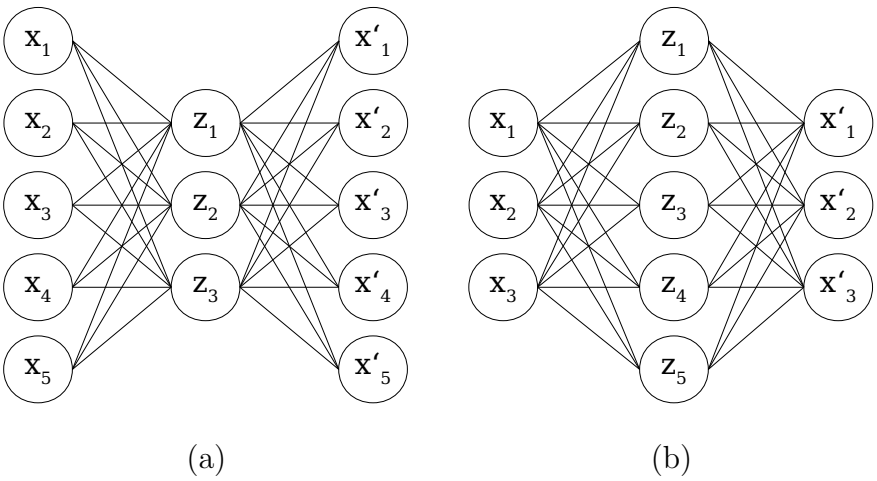


Abb. 3.1. Sparsing Autoencoder, die Sparsity erzeugen durch (a) eine geringere Anzahl von Neuronen in der verdeckten Schicht, oder (b) Sparsity Constraints, die Neuronen erzwingen inaktiv zu bleiben

Eine weitere Variante ist es, dies durch die in Kapitel 2.2 eingeführten *Sparsity Constraints* mit Hilfe eines übervollständigen AE, wie in Abbildung 3.1(b), zu erreichen (nach [Ng17]). Eine Mischung aus beiden Constraints ist ebenfalls möglich, was zu einer erhöhten Kompression führt (nach [Gal16]).

Es ist leicht einzusehen, dass ein Sparsing Autoencoder der Definition und den Grundlagen eines generellen AE aus Kapitel 2 ähnelt. Dies ist der Fall, da diese Art von AE (wie es scheint) als "Normalfall" angesehen wird. Also kann ein AE generell auch als Sparsing Autoencoder bezeichnet werden, sofern keine weiteren Eigenschaften gegeben sind.

3.2 Denoising Autoencoder

Ein *Denoising Autoencoder* (im Folgenden *DAE* genannt) ist eine Variante des AE, die nicht die Eingabe x selbst, sondern eine korrupte Version $\tilde{x}$ von x rekonstruiert (nach [Gal16]).

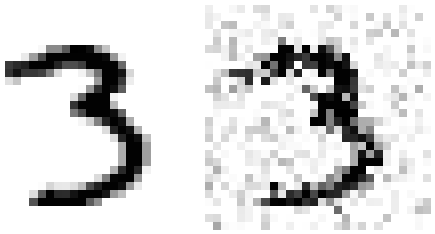
Der Hauptunterschied zur Definition eines generellen AE liegt, nach [Gal16], in zwei Merkmalen:

- Eingabekorruption (engl. *input corruption*)
- Verlustfunktion

Die Korruption der Eingabedaten kann - besonders bei Bildern - Rauschen, fehlende Teile der Daten oder jegliche andere Art der Veränderung sein. Es kommt auf die Art der Daten an, welche Korruption am besten funktioniert. Dabei gilt für $\tilde{x}$:

$$\tilde{x} = corrupt(x)$$

Dies kann gut mit Bildern als Eingabe veranschaulicht werden. In Abbildung 3.2 wurde ein Bild der handgeschriebenen Ziffer 3 aus dem MNIST Datensatz korrumpiert, indem einzelne Pixel verschoben und allgemeines Rauschen mit Hilfe des HSV-Noise hinzugefügt wurde.



Original Korrupt

Abb. 3.2. Ein Bild aus dem MNIST Datensatz und dessen korrumpierte Version (links das Original und rechts die korrumpierte Version)

Die Verlustfunktion kann auch bei einem DAE eine beliebige sein. Der Unterschied besteht, nach [Gal16], darin, dass die Reihenfolge, in der der DAE die

Eingabedaten verarbeitet, ein wenig anders aussieht. Es wurde die Eingabekorruption eingeführt und diese muss nun auch von der Verlustfunktion beachtet werden. Für den Verlust eines DAE gilt also Folgendes:

$$L(x, x') \text{ für } x' = decode(encode(corrupt(x)))$$

Dies bedeutet, nach [Gal16], dass die Ausgabe x' nicht mit der korrumpierten Eingabe $\tilde{x}$ verglichen wird, sondern mit der Originaleingabe x, obwohl x' dennoch aus $\tilde{x}$ erzeugt wurde und nicht aus x. Das hat zur Folge, dass der DAE ein robusteres Verhalten erlernt, also die latente Darstellung in der verdeckten Schicht auch eine korrupte und verrauschte Eingabe zuverlässig rekonstruieren kann.

3.3 Variational Autoencoder

Ein *Variational Autoencoder* (im Folgenden *VAE* genannt) ist eine etwas modernere Variante des AE. Sie ist auch ein wenig praktischer als andere Arten, denn AE werden in der Regel nur selten wirklich praktisch eingesetzt und erzielen meist keine oder nicht signifikant bessere Ergebnisse als herkömmliche Methoden (nach [Cho16]).

Der Name des VAE verrät bereits seine Funktion. Er encodiert und decodiert Eingaben nicht einfach nur, sondern er variiert sie. Zwar gibt es bei anderen AE auch gewisse Abweichungen, aber bei einem VAE sind genau diese Abweichungen das Ziel welches sie erreichen wollen. Dies bedeutet, dass er gezielt verschieden große Abweichungen generieren kann, um zum Beispiel ganz neue Daten zu erstellen. Also ist ein VAE ein *generatives Modell* (nach [Cho16] und [Fra16]). Diese Fähigkeit erlernt er dadurch, dass er, genau wie bei einem generellen AE, eine latente Darstellung der Eingabedaten erstellt. Nur tut ein VAE dies, nach [Fra16], mit einem zusätzlichen Constraint: Die latente Darstellung muss der Gaußschen Normalverteilung (engl. *guassian normal distribution*) ähneln. Die Gleichheit von Wahrscheinlichkeitsverteilungen kann bekanntlich durch die Kullback-Leibler-Divergenz bestimmt werden. Aber anders als bei anderen AE Varianten, wird hier die KL-Divergenz nicht als Regularisierer im klassischen Sinne verwendet, sondern um den latenten Verlust zwischen der latenten Darstellung und der Gaußschen Normalverteilung zu bestimmen (nach [Fra16]). Die Berechnung des Gesamtverlustes sieht, nach [Fra16], also wie folgt aus:

$$L_{VAE} = L(x, x') + KL(z, z_{NV})$$

Dabei ist z_{NV} die erwünschte Normalverteilung aus z. $L(x, x')$ stellt den Verlust aus Eingabe und Ausgabe dar und $KL(z, z_{NV})$ den latenten Verlust. Um dies nun informationstechnisch besser umsetzen zu können, werden einige Parameter umgeformt. Anstatt eine latente Darstellung z aus x zu generieren, wird diese nun in einen Vektor aus Durchschnittswerten (engl. *mean*) z_{mean} und Standardabweichungen (engl. *standard deviation*) z_{stddev} aufgeteilt, wie in Abbildung 3.3 veranschaulicht wird. Dies kann, nach [Fra16] (die Berechnung nach [Cho16] verwendet

andere Parameter, verfolgt allerdings das gleiche Schema), nun genutzt werden, um die KL-Divergenz effizienter und leichter mit Hilfe der TensorFlow Bibliothek zu berechnen:

```
1  latent_loss = 0.5 * tf.reduce_sum(tf.square(z_mean) +
2  tf.square(z_stddev) - tf.log(tf.square(z_stddev)) - 1,1)
```

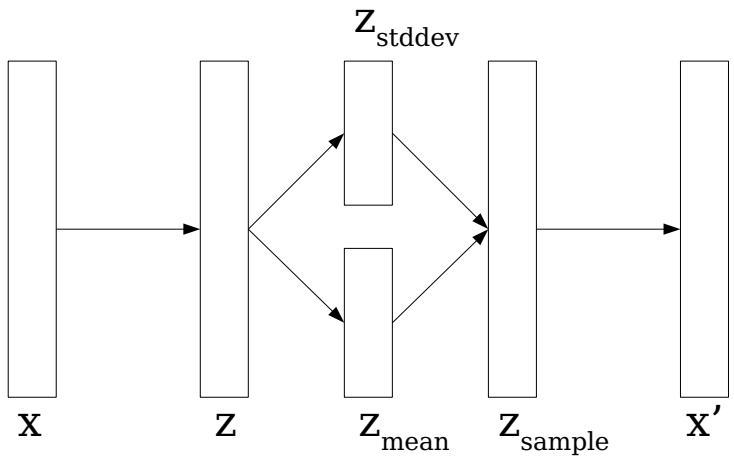


Abb. 3.3. Aufteilung der latenten Darstellung z in Durchschnitte z_{mean} und Standardabweichungen z_{stddev} mit anschließender Generierung eines neuen Musters z_{sample} (Abbildung angelehnt an [Fra16])

Die Aufteilung in zwei Vektoren bietet den Vorteil, dass daraus leicht Muster abgeleitet, also neue Ausgaben generiert werden, können. Das abgeleitete Muster z_{sample} wird, nach [Fra16], dadurch erzeugt, dass z_{stddev} durch eine zufällige Normalverteilung variiert und zu z_{mean} hinzuaddiert wird:

```
1      samples = tf.random_normal([batchsize,n_z],0,1,dtype=tf.float32)
2      z_sample = z_mean + (z_stddev * samples)
```

Durch die Variation der Standardabweichung können so nun verschiedenste Muster generiert werden. Wenn also ein VAE mit dem MNIST Datensatz trainiert wurde, kann der VAE folglich Ziffern von 0 bis 9 erzeugen sowie bestimmte Variationen daraus, allerdings keine Bilder, die nicht in irgendeiner Weise aus den Originaldaten ableitbar sind. Dies kann gut graphisch dargestellt werden, indem ein festes Intervall verwendet und Zahlen daraus in einem bestimmten Abstand zu der Standardabweichung hinzumultipliziert wird, wie in Abbildung 3.4 zu sehen ist. Aus einem VAE können, nach [Cho16], insgesamt drei Modelle instantiiert werden:

- Ein Ende-zu-Ende AE, welcher die Eingabe rekonstruiert
- Ein AE, welcher die Eingabe auf die latente Darstellung abbildet
- Ein Generator, welcher aus der latenten Darstellung entsprechende Muster rekonstruiert

Die Instantiierung als Generator benötigt nur den Decoder, da die latente Darstellung als solche nicht verwendet wird, was im Grunde das Gegenstück zum Sparsing Autoencoder darstellt, da dieser hauptsächlich den Encoder für seine Funktion benötigt.

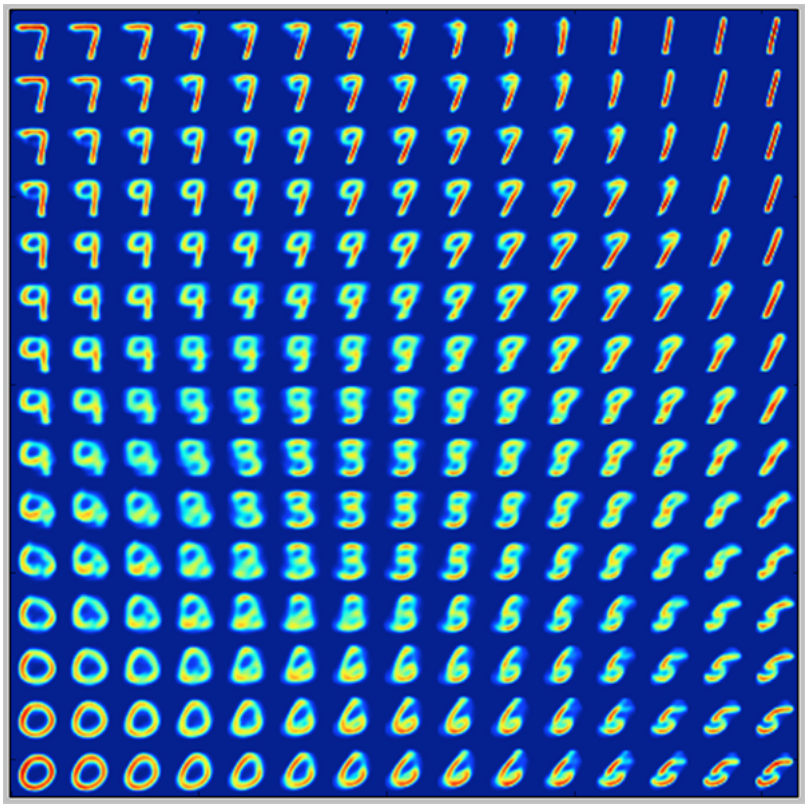


Abb. 3.4. Ausgabe nach der Variation der Standardabweichung eines mit dem MNIST Datensatz trainierten VAE (Abbildung von [Cho16])

3.4 Contractive Autoencoder

Ein Contractive Autoencoder (im Folgenden *CAE* genannt), ist ein AE, der kleine Abweichungen der Eingabedaten kompensiert, also ähnlich wie die Fähigkeit eines DAE. Aber anders als bei einem DAE bezieht sich dies ausschließlich auf die Trainingsdatenmenge, also müssen diese nicht vorher korrumpiert werden (nach [JS17]).

Ähnlich wie bei anderen Variationen eines AE wird, nach [Kri16], die Verlustfunktion so angepasst, dass eine entsprechende Bestrafung erfolgt. In diesem Fall ist der zusätzliche Penalty-Term die Jacobi-Matrix innerhalb der Frobenius-Norm aus dem Eingabevektor:

$$||J_h(x)||_F^2 = \sum_{ij} \left(\frac{\partial h_j(x)}{\partial x_i} \right)^2$$

h_j ist dabei das j-te Neuron in der verdeckten Schicht, oder präziser, der Aktivierungswert dieses Neurons. Die Jacobische wird verwendet, da hier auf einem Vektor partiell abgeleitet wird. Diese Ableitung hat, nach [JS17], den Effekt, dass ein Anstieg des Aktivierungswertes h_j sich auch auf die Jacobische auswirkt, und somit die latente Repräsentation bestraft wird. Dieses Verhalten erinnert an das Sparsity Constraint eines Sparsing Autoencoders. Ein CAE verwendet als Aktivierungsfunktion meist die Sigmoid-Funktion, welche sehr flach verläuft, sodass sich die Neuronen hauptsächlich auf der linken Seite der Funktion befinden würde. Dies bedeutet, dass Neuronen zu Anfang geringe Ableitungswerte haben, welche zu einer einer kleinen Jacobi-Matrix führen (nach [JS17]).

Wie schon erwähnt, ähneln sich CAEs und DAEs in ihren Fähigkeiten, allerdings verteilt sich diese bei einem DAE auf den Encoder und den Decoder, da das Ergebnis beider Teilnetze, also x' mit x verglichen wird. Bei einem CAE wiederum bezieht sich diese Fähigkeit nur auf den Encoder, da der Penalty-Term nur für den Encoder, also die Verbindung zwischen Eingabe- und verdeckter Schicht, gilt und die Eingabedaten nicht verändert werden. Dies muss in Betracht gezogen werden, abhängig davon, welche Art von Daten verarbeitet wird und ob es auf die latente Repräsentation oder die Rekonstruktion ankommt (nach [JS17]).

4

Anwendungsgebiete

Die grundlegende Idee eines AE wirkt auf den ersten Blick ein wenig "unbrauchbar", da das Prinzip lautet, eine möglichst gleiche Ausgabe zu einer gegebenen Eingabe zu generieren. Aber genau dieses "unbrauchbare" Verhalten kann in der Tat für viele Gebiete sehr nützlich sein.

Eine der Hauptanwendungsgebiete ist die Dimensionsreduktion, zum Beipsiel bei der Verkleinerung von Bildmaterial. Dafür ist besonders der Sparsing Autoencoder aus Kapitel 3.1 geeignet. Dieser kann durch geeignete Sparsity Constraints eine latente Darstellung der Eingabe erzeugen, die dann bei Bildern eine verkleinerte Version des Ursprungsbildes zeigt. Dies kann zum Beispiel bei der Archivierung von Bildmaterial genutzt werden um Speicherplatz zu sparen (wenn die Bilder wieder benötigt werden, besteht die Möglichkeit, sie mit Hilfe des Decoders wieder zu rekonstruieren). Bei dieser Art der Verwendung wird für die eigentliche Dimensionsreduktion nur der Encoder, also die Eingabe- und verdeckte Schicht verwendet. Erst zur Rekonstruktion (falls dies überhaupt stattfindet) wird der Decoder zur Verwendung gezogen.

Ein weiteres Anwendungsgebiet wäre die statistische Analyse von Distributionen. Bei dieser Anwendung werden, ähnlich wie bei dem statischen Verfahren der Hauptkomponentenanalyse (*PCA*), die wichtigsten und signifikantesten Aspekte der Eingabe gefiltert. Dies kann dabei helfen, das Kaufverhalten von Kunden zu analysieren und herauszufinden, warum oder von welchen Personen bestimmte Artikel gekauft werden. Bei dieser Anwendung hängt die Dimension der Eingabe von der Anzahl der Parameter ab, die diese Verteilung repräsentieren (z. B. Alter, Beruf, Familienstand, Geschlecht, usw. des Käufers). Die Dimension der verdeckten Schicht leitet sich davon ab, auf wie viele Parameter sich beschränkt werden soll, die vermutlich zum Kauf geführt haben.

Daten müssen nicht immer perfekt sein und können auch Unstimmigkeiten aufweisen, die aufgrund von unsauberen Messungen oder Datenkorruption entstehen. Ein Denoising Autoencoder, der in Kapitel 3.2 eingeführt wurde, nimmt sich genau diesem Problem an. Er ist darauf ausgelegt, gestörte oder kaputte Daten dennoch zu rekonstruieren und die Korrumpierungen möglichst gut zu kompensieren. Dies kann beispielsweise auf korrupte Bilder angewendet werden. Auch wenn ein ganzer Teil des Bildes fehlen sollte, kann ein entsprechend trainierter Denoising Autoencoder den fehlenden Teil des Bildes ersetzen. Also funktioniert dieses Verfahren

auch, wenn generell Teile von Datensätzen verloren gegangen sind. Dass verlorene Daten bis zu einem gewissen Grad nicht perfekt rekonstruiert werden können, sollte allerdings klar sein. Der Denoising Autoencoder kann aber auch zusammen mit Sparsity Constraints verwendet werden, sodass eine "Fehlerkorrektur" auch bei der Dimensionsreduktion verwendet werden kann, um ein robusteres Verhalten zu erreichen. Sollen allerdings nur kleinere Änderungen der Daten berücksichtigt werden, würde auch ein Contractive Autoencoder aus Kapitel 3.4 ähnliche Ergebnisse liefern.

Eine der interessantesten Gebiete der künstlichen Intelligenz ist die *kreative* Erzeugung von Bildern oder Musik. Ein Variational Autoencoder aus Kapitel 3.3 kann als generatives Modell instantiiert werden und somit neue, auf den Trainingsdaten basierende, Ausgaben erzeugen. Wird ein VAE mit Bildern von Gesichtern trainiert, ist er in der Lage auch Gesichter zu generieren, die noch nicht existieren. Auch neue Musikstücke könnte er komponieren. In dieser Hinsicht ist der VAE möglicherweise der interessanteste und wichtigste Autoencoder.

Literaturverzeichnis

Caw03. Cawsey, Alison: *Kunstlicke Intelligenz Im Klartext.* Pearson Studium, 2003.

Cho16. Chollet, Francois: *Building Autoencoders in Keras.* https://blog.keras.io/building-autoencoders-in-keras.html, 04 2016.

Cri17. Cristani, Marco: *Deep Neural Networks.* http://www.di.univr.it/documenti/OccorrenzaIns/matdid/matdid110761.pdf, 2017.

Fra16. Frans, Kevin: *Variational Autoencoders Explained.* http://kvfrans.com/variational-autoencoders-explained/, 08 2016.

Gal16. Galeone, P.: *Introduction to Autoencoders.* https://pgaleone.eu/neural-networks/2016/11/18/introduction-to-autoencoders/, 2016.

IG16. Ian Goodfellow, Yoshua Bengio, Aaron Courville: *Deep Learning.* http://www.deeplearningbook.org, 2016.

JS17. Jordan Spooner, Qiang Feng, u. A.: *Contractive Autoencoders.* https://www.doc.ic.ac.uk/∼js4416/163/website/autoencoders/contractive.html, 11 2017.

Kri16. Kristiadi, Augustinus: *Deriving Contractive Autoencoder and Implementing it in Keras.* https://wiseodd.github.io/techblog/2016/12/05/contractive-autoencoder/, 12 2016.

Ng17. Ng, Andrew: *Sparse Autoencoders.* https://web.stanford.edu/class/cs294a/sparseAutoencoder.pdf, 2017.

PN12. Peter Norvig, Stuart Russell: *Künstliche Intelligenz.* Pearson Studium, 2012.

vA04. Alphen, Debbie van: *Introduction to Neural Networks.* http://www.csun.edu/∼skatz/nn_proj/intro_nn.pdf, 2004.

Wei. Weisstein, Eric W.: *Hyperbolic Tangent.* http://mathworld.wolfram.com/HyperbolicTangent.html.