Hochschule München University of Applied Sciences

Hauptkomponentenanalyse

(Principal Component Analysis, PCA)

vs.

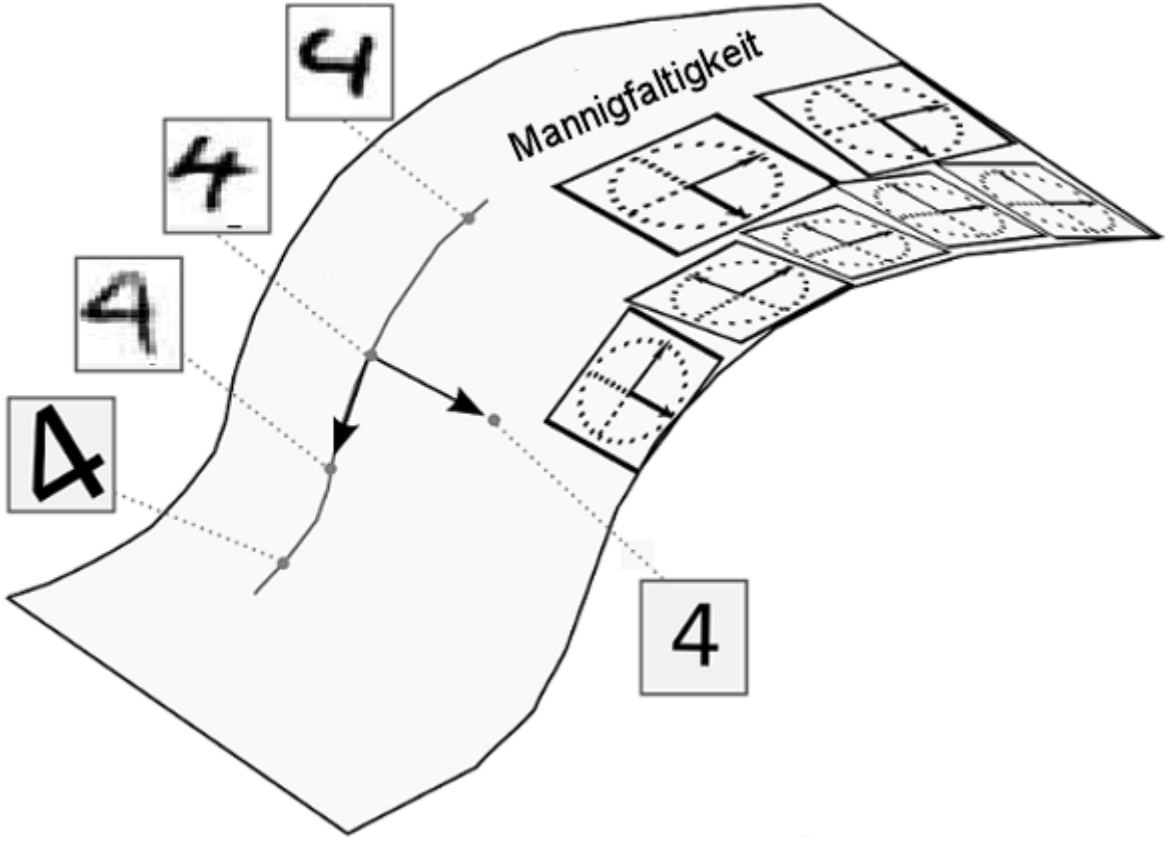
Denoising Variational Autoencoders

an Hand von Beispielen

Dr. Cristian Axenie

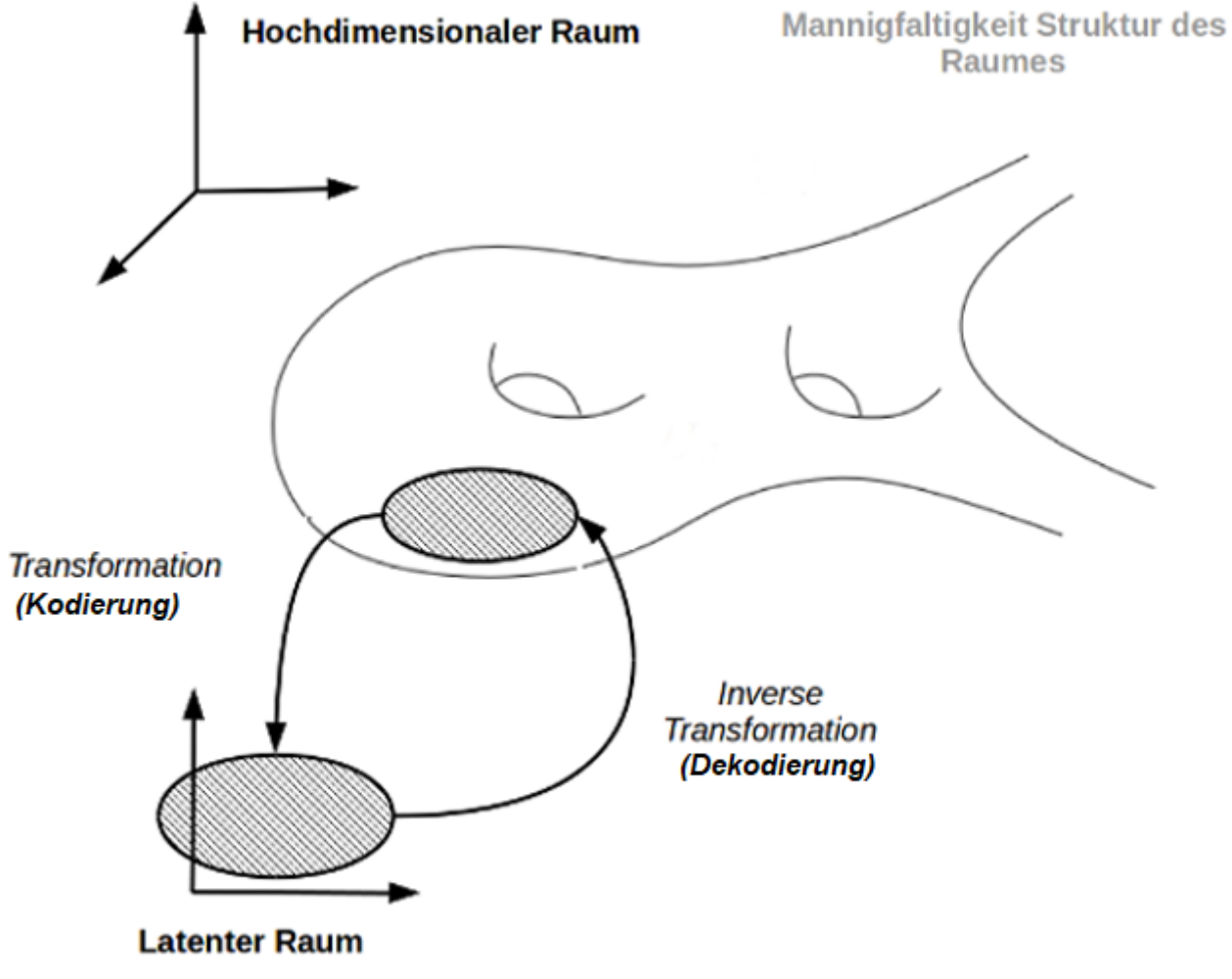
Eine intuitive Perspektive ...

"... realistische, hochdimensionale Daten konzentrieren sich in der Nähe einer nichtlinearen, niedrigdimensionalen Mannigfaltigkeit ..." [Lei et al., 2018]



Eine intuitive Perspektive ...

Aber wie lernt man die Mannigfaltigkeit und die Wahrscheinlichkeitsverteilung darauf?



PCA vs. DVAE an Hand von Beispielen

PCA vs. DVAE an Hand von Beispielen

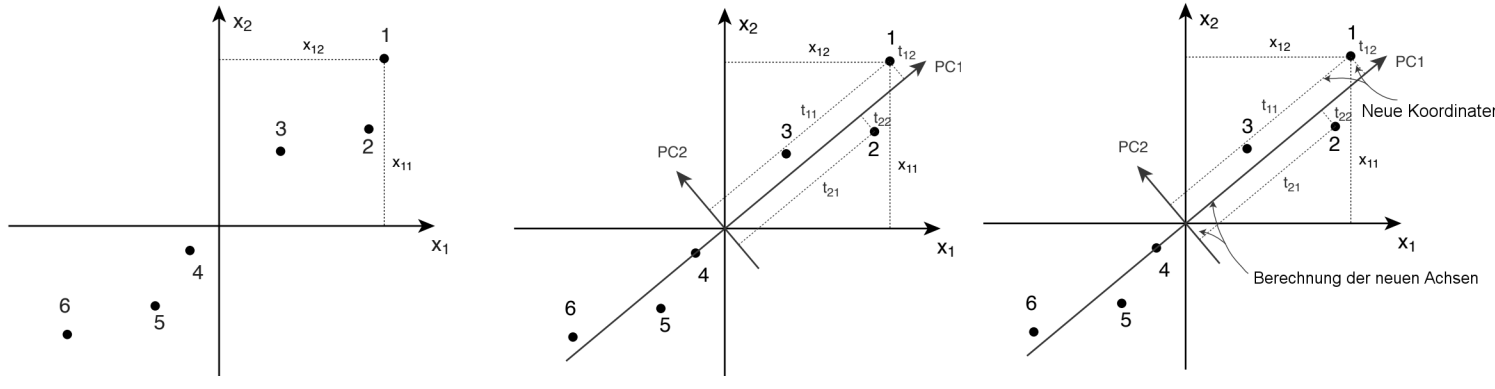
Der **MNIST (Modified National Institute of Standards and Technology) Datensatz** von handgeschriebenen Zahlen besteht aus **60,000 Trainings- und 10,000 Test-Beispielen**. Die Zahlen wurden hinsichtlich Ihrer Größe **normalisiert und in einem Bild fester Größe zentriert**.

Vorstellung der Wettbewerber

PCA

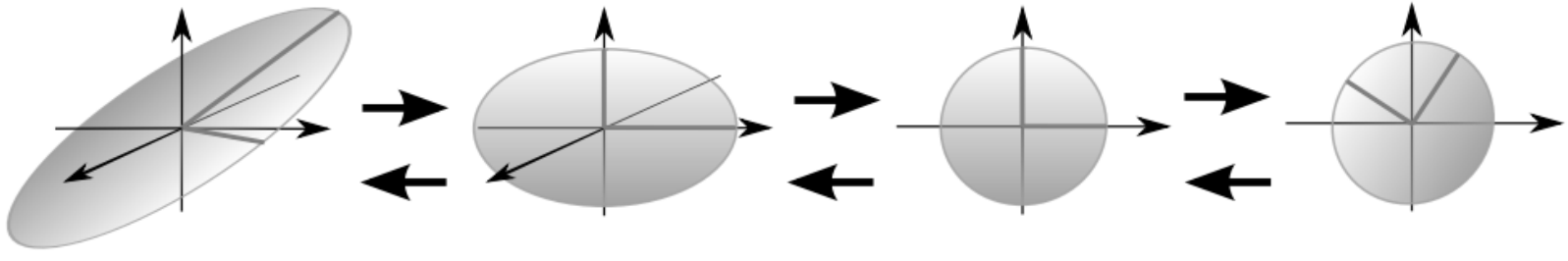
PCA

- **Unüberwachtes** Lernen
- **Lineare Transformation**



- **"Transformiere"** eine Menge von Beobachtungen in ein **anderes Koordinatensystem**, in dem die Werte der ersten Koordinate (Komponente) die **größtmögliche Varianz** aufweisen [Friedman et al., 2017]
- Die **resultierenden Koordinaten (Komponenten)** sind **nicht** mit den ursprünglichen Koordinaten **korreliert**

PCA



- Wird zur **Dimensions-Reduzierung** genutzt (Komprimierung)
- Die **Rekonstruktion der Beobachtungen**("decoding") aus den führenden **Hauptkomponenten** hat den **niedrigsten quadratischen Fehler**

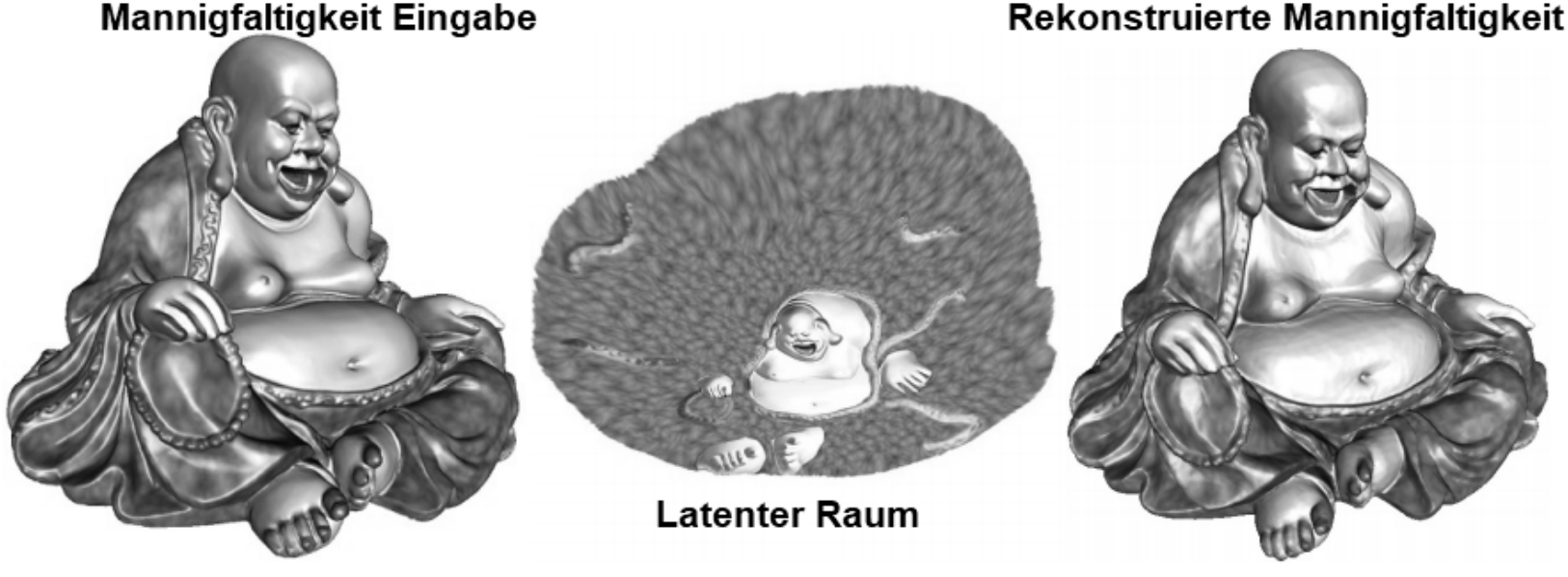
Autoencoders

Autoencoders

- unüberwachtes **neuronales Netz**
- **minimiert** den Fehler zwischen Rekonstruktionen und Beobachtungen [Goodfellow et al., 2016]
- lernt die **Identitätsfunktion**
- wird mit Hilfe von **Fehlerrückführung (Backpropagation) trainiert**
- aufgetrennt um **Kodierung und Dekodierung**

Autoencoders

Das folgende Schaubild zeigt eine typische **Autoencoder Pipeline**



PCA vs. Autoencoders

Implementierung

In [ ]:

```
# we use Keras to implement, layer-by-layer the DVAE and PCA
from keras.layers import Input, Dense, Lambda
from keras.models import Model
from keras import backend as K
from keras import metrics
from keras.datasets import mnist
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Conv2DTranspose,Reshape
from sklearn.decomposition import PCA
```

Experiment Parametrierung

In [ ]:

```
# training params for PCA vs. DVAE
num_train = 50000
n_images = 6
batch_size = 205
original_dim = 784
latent_dim = 8
epochs = 1000
epsilon_std = 1.0
noise_factor = 0.5
```

In [ ]:

```
# get the MNIST digits
(x_train, y_train), (x_test, y_test) = mnist.load_data()
# prepare data for PCA
# training
x_test_pca = x_test
shape_x_test = x_test_pca.shape
pcaInputTest = np.reshape(x_test,[shape_x_test[0],shape_x_test[1]*shape_x_test[2]]).asty
pe('float32')/255
# prepare data for DVAE
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), 28,28,1))
x_test = x_test.reshape((len(x_test), 28,28,1))
noise_train = x_train + noise_factor * np.random.randn(*x_train.shape)
noise_test = x_test + noise_factor * np.random.randn(*x_test.shape)
# clip the images to be between 0 and 1
noise_train = np.clip(noise_train, 0., 1.)
noise_test = np.clip(noise_test, 0., 1.)
```


0
10
20
30
40
50
0
20
40
60
80
100
120
140
160

Grundlegende Mathematik der PCA

Lineare Transformation

Es sei $\{y_i\}_{i=1}^{N}$ eine Menge von N Beobachtungs-Vektoren der Dimension n mit $n \leq N$.

Eine **lineare Transformation** eines **endlich-dimensionalen** Vektors kann als **Matrix Multiplikation** ausgedrückt werden:

$$x_i = W y_i$$

mit $y_i \in R^n, x_i \in R^m$ und $W \in R^{nxm}$.

Lineare Transformation

- Das $j-te$ Element in x_i ist das **Innere Produkt** von y_i und der $j-ten$ Spalte der Matrix W, welche wir durch w_j bezeichen. Es sei $Y \in R^{nxN}$ die Matrix, welche wir durch horizontale Aneinanderreihung der Vektoren $\{y_i\}_{i=1}^{N}$ erhalten,

$$Y = \begin{bmatrix} |\dots| \\ y_1 \dots y_N \\ |\dots| \end{bmatrix}$$

- Aus der **linearen Transformation** folgt:

$$X = W^T Y, X_0 = W^T Y_0,$$

wobei Y_0 die **Matrix der zentrierten Elemente** (d.h. wir subtrahieren den Mittelwert von jeder Beobachtung) bezeichnet, und **Kovarianzmatrix** $Y_0 {Y_0}^T$.

Dimensionsreduzierung, Komprimierung

PCA wird zur **Dimensions-Reduktion** verwendet, da sie durch die durch eine lineare Transformation die **Anzahl der Variablen reduziert**.

Da nur die ersten m **Hauptkomponenten erhalten** werden, **verliert** PCA **information** (d.h. **verlustreiche Komprimierung**).

Der **Verlust** (*Summe des quadratischen Rekonstruktions-Fehlers*) wird jedoch durch die **Maximierung der Komponenten-Varianzen minimiert**

$$\min_{W \in R^{nxm}} \|Y_0 - WW^T Y_0\|_F^2, W^T W = I_{mxm}$$

wobei F die Frobenius-Norm bezeichnet.

Skalierung

Zur Berechung der PCA können viele verschiedene **iterative Algorithmen** eingesetzt werden

- QR Algorithmen
- Jacobi Algorithmus
- Power methode
- Singulärwert-Zerlegung (Singular Value Decomposition, SVD)

Für **sehr große Datenmengen** eignen sich diese Algorithmen **nicht**!

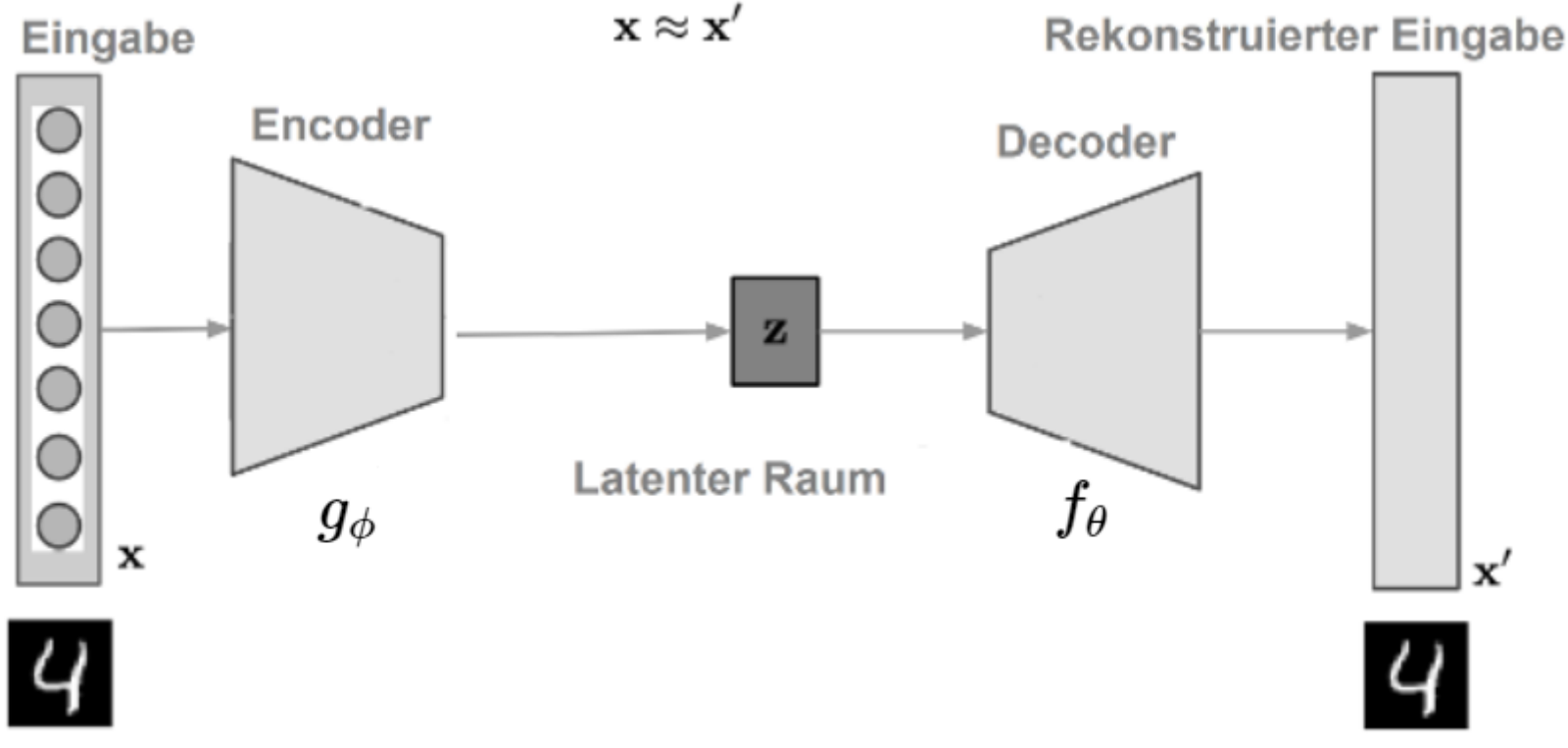
In [ ]:

```python
# analytical PCA of the training set
def analytical_pca(y):
    # variance to explain
    pca = PCA(0.7)
    # apply PCA
    pca.fit(y)
    # extract the components
    loadings = pca.components_
    # apply the transformation
    components = pca.transform(y)
    # reconstruct from components for visualization
    filtered = pca.inverse_transform(components)
    return filtered
```

Grundlegende Mathematik der Autoencoder

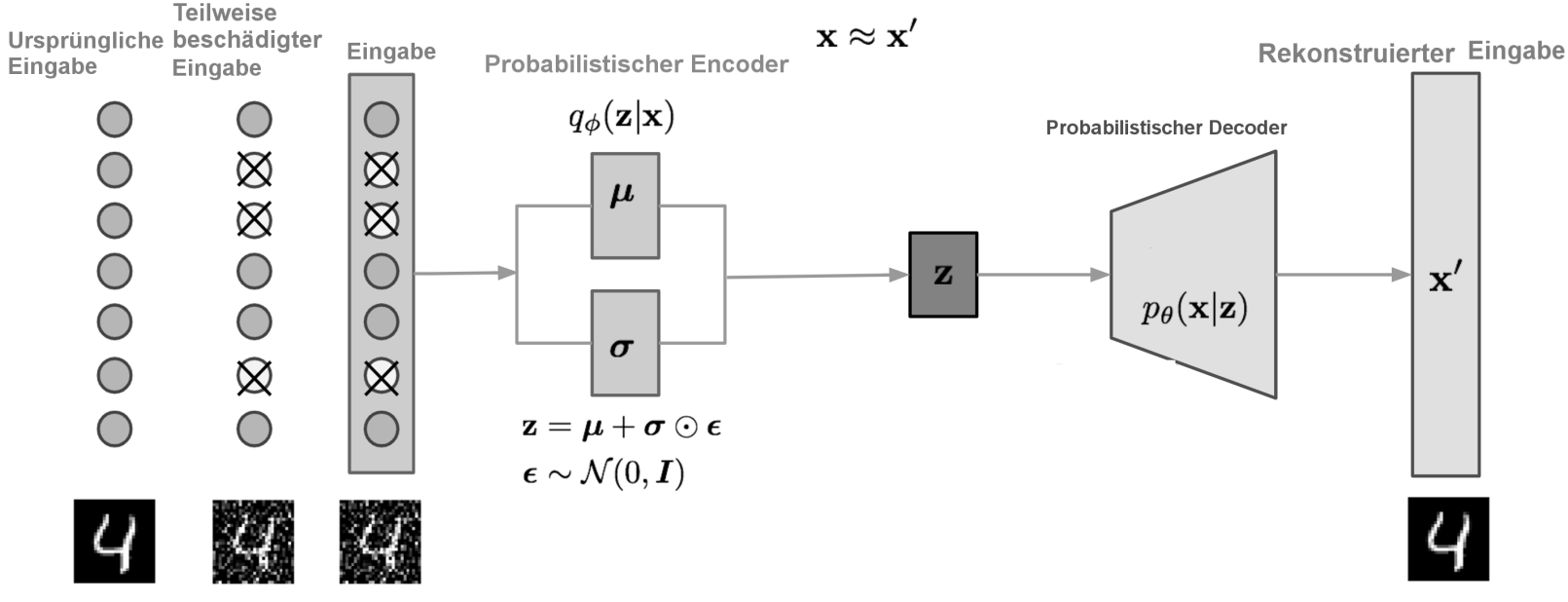
Für jeden Eingangsvektor x der Dimension d des kompletten Datensaztes der Länge n generiert das neuronale Netz eine Rekonstruktion x' durch:

- **Kodierung der Eingangsdaten** (d.h. verwende die lineare / nicht-lineare Transformation $g_\phi(.)$)
- dies liefert eine **komprimierte Kodierung** in der dünnsten Netzwerk-Ebene, z
- **Dekodierung der komprimierten Eingangsdaten** durch Anwendung der linearen / nicht-linearen Transformation $f_\theta(.)$



Denoising Variational Autoencoders (DVAE)

Das Funktionsprinzip **unterscheidet sich** vom grundlegenden Autoencoder dahingehend, dass ein gewisses Maß an **Störrauschen** (einer **gewissen Wahrscheinlichkeitsverteilung** folgend) den **Eingangsdaten hinzugefügt wird** und dass die **verborgenen Ebenen** dieses Rauschen **ausgleichen muss** um die Eingangsdaten zu **rekonstruieren** [Im, Bengio et al., 2017, Kingma et al., 2017].



In [ ]:

```
# Implement the DVAE
# encoder part
x_noise = Input(shape=(28,28,1))
conv_1 = Conv2D(64,(3, 3), padding='valid',activation='relu')(x_noise)
conv_2 = Conv2D(64,(3, 3), padding='valid',activation='relu')(conv_1)
pool_1 = MaxPooling2D((2, 2))(conv_2)
conv_3 = Conv2D(32,(3, 3), padding='valid',activation='relu')(pool_1)
pool_2 = MaxPooling2D((2, 2))(conv_3)
h=Flatten()(pool_2)
z_mean = Dense(latent_dim)(h)
z_log_var = Dense(latent_dim)(h)
```

In [ ]:

```
# Implement the DVAE
# decoder part
# we instantiate these layers separately so as to reuse them later
z=Reshape([1,1,latent_dim])(z)
conv_0T = Conv2DTranspose(128,(1, 1), padding='valid',activation='relu')(z)#1*1
conv_1T = Conv2DTranspose(64,(3, 3), padding='valid',activation='relu')(conv_0T)#3*3
conv_2T = Conv2DTranspose(64,(3, 3), padding='valid',activation='relu')(conv_1T)#5*5
conv_3T = Conv2DTranspose(48,(3, 3), strides=(2, 2),padding='same',activation='relu')(co
nv_2T)#10*10
conv_4T = Conv2DTranspose(48,(3, 3), padding='valid',activation='relu')(conv_3T)#12*12
conv_5T = Conv2DTranspose(32,(3, 3), strides=(2, 2),padding='same',activation='relu')(co
nv_4T)#24*24
conv_6T = Conv2DTranspose(16,(3, 3), padding='valid',activation='relu')(conv_5T)#26*26
x_out = Conv2DTranspose(1,(3, 3), padding='valid',activation='sigmoid')(conv_6T)#28*28
```

DVAE

- DVAE **Verlustfunktion** beinhaltet die Erstellung von Beispielen aus $z \backsim q_\phi(z|x)$. Dies ist ein **stochastischer Prozess** und eignet sich daher **nicht zur Fehlerrückführung**.

- Die **geschätzte Posteriori-Verteilung $\boldsymbol{q_\phi(z|x)}$** approximiert die tatsächliche Verteilung $p_\theta(z|x)$.

- Wir können die **Kullback-Leibler Abweichung**, D_{KL} benutzen um die **Differenz der beiden Verteilungen** zu quantifizieren.

In [ ]:

```
# Implement the DVAE
# reparameterization trick
def sampling(args):
    z_mean, z_log_var = args
    epsilon = K.random_normal(shape=(K.shape(z_mean)[0], latent_dim), mean=0.,
                              stddev=epsilon_std)
    return z_mean + K.exp(z_log_var / 2) * epsilon

z = Lambda(sampling, output_shape=(latent_dim,))([z_mean, z_log_var])
```

DVAE

Durch **Minimierung des Verlusts**, **maximieren** wir daher die **untere Schranke der Wahrscheinlichkeit** (**evidence lower bound (ELBO)**) zur Generierung echter Daten-Beispiele.

In [ ]:

```
# Implement the DVAE
# instantiate model
dvae = Model(x_noise, x_out)
dvae.summary()

# Compute loss
def DVAE_loss(x_origin,x_out):
    x_origin=K.flatten(x_origin)
    x_out=K.flatten(x_out)
    xent_loss = original_dim * metrics.binary_crossentropy(x_origin, x_out)
    kl_loss = - 0.5 * K.sum(1 + z_log_var - K.square(z_mean) - K.exp(z_log_var), axis=-1
)
    dvae_loss = K.mean(xent_loss + kl_loss)
    return dvae_loss

# compile the model
dvae.compile(optimizer='adam', loss=DVAE_loss)
```

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 28, 28, 1)	0	
conv2d_1 (Conv2D)	(None, 26, 26, 64)	640	input_1[0][0]
conv2d_2 (Conv2D)	(None, 24, 24, 64)	36928	conv2d_1[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 12, 12, 64)	0	conv2d_2[0][0]
conv2d_3 (Conv2D)	(None, 10, 10, 32)	18464	max_pooling2d_1[0][0]
max_pooling2d_2 (MaxPooling2D)	(None, 5, 5, 32)	0	conv2d_3[0][0]
flatten_1 (Flatten)	(None, 800)	0	max_pooling2d_2[0][0]
dense_1 (Dense)	(None, 8)	6408	flatten_1[0][0]
dense_2 (Dense)	(None, 8)	6408	flatten_1[0][0]
lambda_1 (Lambda)	(None, 8)	0	dense_1[0][0] dense_2[0][0]
reshape_1 (Reshape)	(None, 1, 1, 8)	0	lambda_1[0][0]
conv2d_transpose_1 (Conv2DTrans	(None, 1, 1, 128)	1152	reshape_1[0][0]
conv2d_transpose_2 (Conv2DTrans	(None, 3, 3, 64)	73792	conv2d_transpose_1[0][0]
conv2d_transpose_3 (Conv2DTrans	(None, 5, 5, 64)	36928	conv2d_transpose_2[0][0]
conv2d_transpose_4 (Conv2DTrans	(None, 10, 10, 48)	27696	conv2d_transpose_3[0][0]
conv2d_transpose_5 (Conv2DTrans	(None, 12, 12, 48)	20784	conv2d_transpose_4[0][0]
conv2d_transpose_6 (Conv2DTrans	(None, 24, 24, 32)	13856	conv2d_transpose_5[0][0]
conv2d_transpose_7 (Conv2DTrans	(None, 26, 26, 16)	4624	conv2d_transpose_6[0][0]
conv2d_transpose_8 (Conv2DTrans	(None, 28, 28, 1)	145	conv2d_transpose_7[0][0]

Total params: 247,825
Trainable params: 247,825

In [ ]:

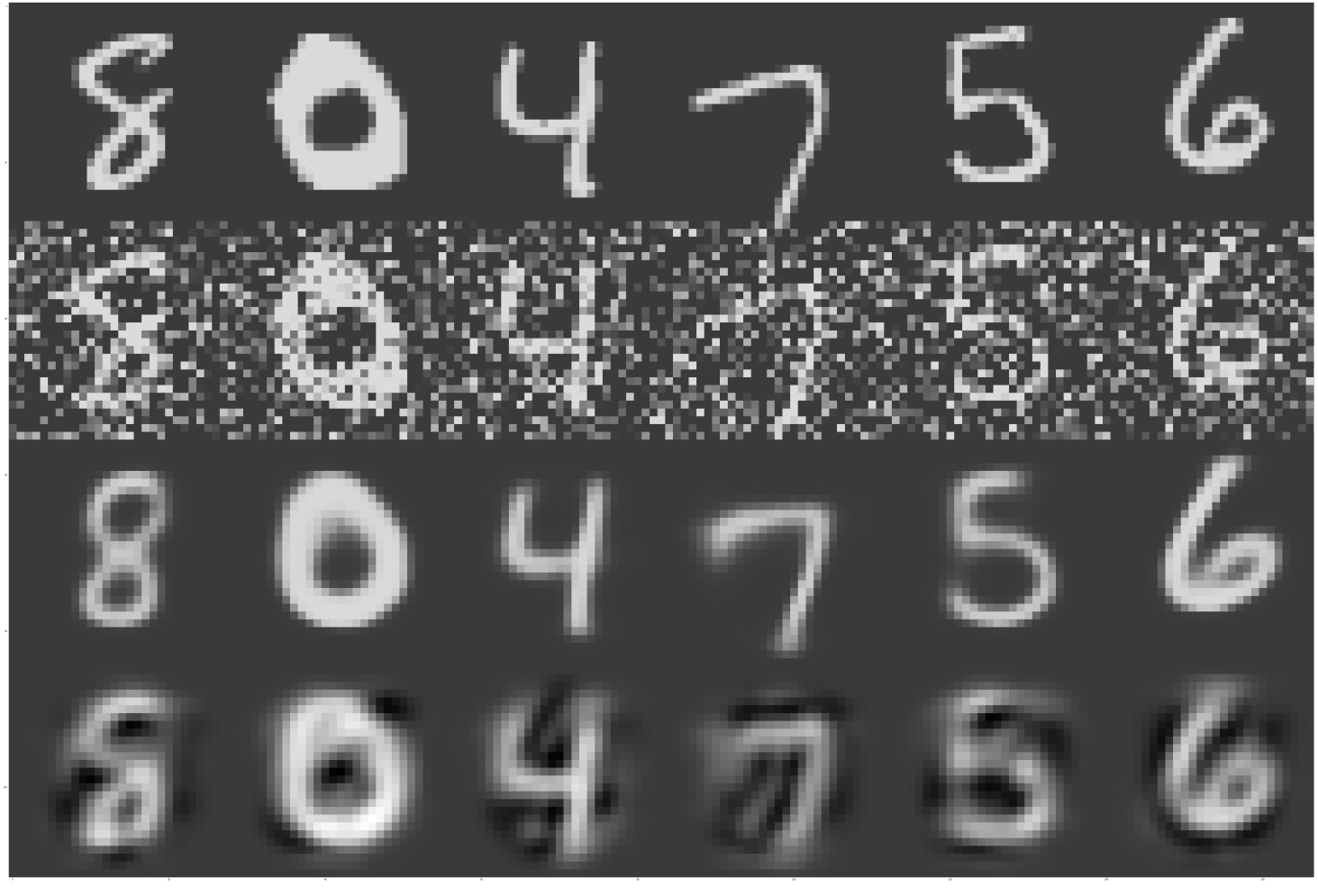
```
# Train the DVAE
dvae.fit(noise_train,x_train,  shuffle=True, epochs=epochs, batch_size=batch_size, valid
ation_data=(noise_test, x_test))

# Comparison PCA vs. DVAE
# testing the DVAE
num_test=10000
showidx=np.random.randint(0,num_test,n_images)
x_out=dvae.predict(x_test[showidx])

# prepare data for testing PCA
pcaInputTest = np.reshape(x_test,[shape_x_test[0],shape_x_test[1]*shape_x_test[2]]).asty
pe('float32')/255
pcaOutput = analytical_pca(pcaInputTest)
```

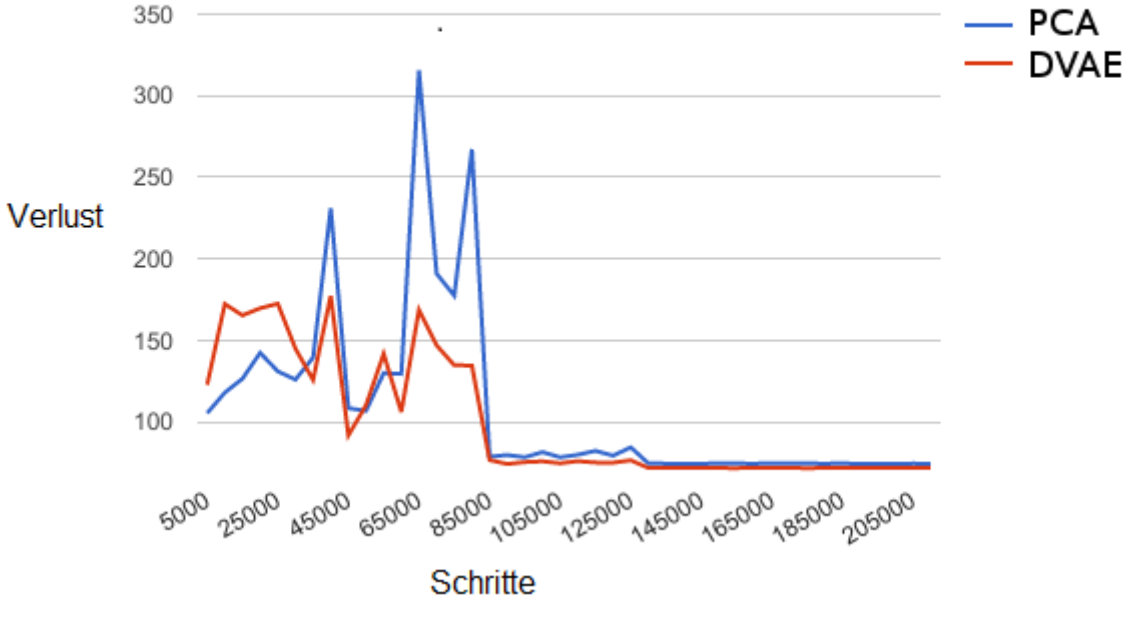
Vergleich von PCA und DVAE

Inferenz



Vergleich von PCA und DVAE

Kostenfunktion



Vergleich zwischen PCA und DVAE

Lernen der Mannigfaltigkeit

PCA	DVAE
Kodierung/Dekodierung, keine Robustheit gegen Rauschen	nicht-linear, probabilistische Kodierung/Dekodierung mit Robustheit gegen Rauschen und nicht-linearen Aktivierungsfunktionen
unkorrelierte Koordinaten	korrelierte Ausgansdaten an der dünnsten Netzwerkebene
Koordinaten sind in absteigener Reihenfolge der Varianz geordnet	Koordinaten sind ungeordnet
die Spalten der Transformations-Matrix sind orthonormal	die Spalten der Transformations-Matrix sind nicht notwendigerweise orthonormal
Robustheit gegenüber moderatem Rauschen mit bekannten Verteilungen	Robustheit gegen eine Vielzahl verschiedener Arten und Größenordnungen an injeziertem Rauschen (masking noise, Gaussian noise, salt-and-pepper noise), da das Entrauschen entscheidung für die Generalisierung ist
einfacher Algorithmus (ohne Regularisierung), geringe Robustheit	die Punkte in niedrig-dimensionalen Mannifaltigkeiten sind robust gegen Rauschen im hoch-dimensionalen Beobachtungs-Raum

Vergleich zwischen PCA und DVAE

Training

PCA	DVAE
Abbildung der Eingangsdaten auf einen festen Vektor	Abbildung der Eingangsdaten auf eine Wahrscheinlichkeitsverteilung
iterative Methoden: QR Zerlegung, Jacobi Algorithmus, Singulärwertzerlegung	Fehlerrückführung (Backpropagation)
aufgrund der Kovarianz-Berechnung ineffizient bei großen Datenmengen	effizient bei großen Datenmengen aufgrund der starken Fähigkeit des Erlernens der Mannigfaltigkeit
basiert auf der Korrelations-/Kovarianz-Matrix, welche - zumindest in der Theorie - sehr empfindlich gegenüber Ausreißern sein kann	kann Beispiele direkt aus dem Eingangsraum generieren und daher die Eigenschfaten des Eingangsrauschens beschreiben ("reparametrization trick")

PCA vs. Autoencoders

”Zwei identische Fremde”

PCA vs. Autoencoders

- Ein **Autoencoder** mit einer einzelnen **voll verbundenen (fully-connected) versteckten Ebene**, einer **linearen Aktivierungsfunktion** und dem **quadratischen Fehler als Kostenfunktion** ist **eng mit der PCA verwandt** - seine **Gewichten** spannen den **Untervektorraum der Hauptkomponenten** auf [Plaut, 2018]

- Bei **Autoencodern** sorgt die **diagonale Approximation beim Kodiervorgang** zusammen mit der **inhärenten Stochastizität** für lokale **Orthogonalität beim Dekodieren** [Rolinek et al, 2019]

Lieraturverzeichnis

[Goodfellow et al., 2016] Ian Goodfellow, Yoshua Bengio and Aaron Courville, Deep Learning, MIT Press, 2016.

[Friedman et al., 2017] Jerome H. Friedman, Robert Tibshirani, and Trevor Hastie, The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Springer, 2017.

[Plaut, 2018] Plaut, E., 2018. From principal subspaces to principal components with linear autoencoders. arXiv preprint arXiv:1804.10253.

[Im, Bengio et al., 2017] Im, D.I.J., Ahn, S., Memisevic, R. and Bengio, Y., 2017, February. Denoising criterion for variational auto-encoding framework. In Thirty-First AAAI Conference on Artificial Intelligence.

[Rolinek et al, 2019] Rolinek, M., Zietlow, D. and Martius, G., 2019. Variational Autoencoders Pursue PCA Directions (by Accident). In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 12406-12415).

[Lei et al., 2018] Lei, N., Luo, Z., Yau, S.T. and Gu, D.X., 2018. Geometric understanding of deep learning. arXiv preprint arXiv:1805.10451.

[Kingma et al., 2013] Kingma, D.P. and Welling, M., 2013. Auto-encoding variational bayes. arXiv preprint arXiv:1312.6114.

Maximale Varianzkomponenten, Kovarianz und Dekorrelation

- Der erste Ladungs-Vektor ist derjenige Einheitsvektor mit dem das innere Produkt der Beobachtungs-Vektoren die **größte Varianz** aufweisen:
$$\max w_1^T Y_0 Y_0^T w_1 \, , w_1^T w_1 \, = 1$$

- Die Lösung der vorherigen leichung ist der erste Eigenvektor der **Kovarianz-Matrix** $Y_0 Y_0^T$, welcher zum größten Eigenwert gehört.

- Die Matrix P kann durch **Diagonalisierung der Kovarianz-Matrix** berechnet werden:

$$Y_0 Y_0^T = P \Lambda P^{-1} = P \Lambda P^T$$

$\Lambda = Y_0 Y_0^T$ ist eine Diagonal-Matrix, deren Diagonal-Elemente $\{\lambda_i\}_{i=1}^N$ der Größe nach absteigend sortiert sind. $Y = PX$ liefert die inverse Tranformation. Da die Kovarianz-Matrix von X diagonal ist, ist die PCA eine **dekorrelierende Transformation**.

Singulärwert-Zerlegung

(Singular Value Decomposition, SVD)

Ein Vektor v der Dimension N ist ein **Eigenvektor** einer quadratischen N × N Matrix A, wenn diese die folgende **lineare Gleichung** erfüllt

$$Av = \lambda v$$

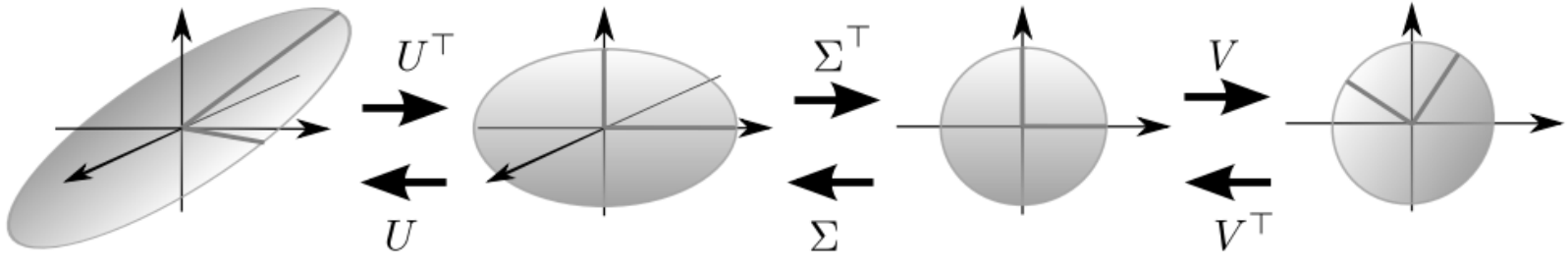
wobei λ ein skalarer Wert ist, welcher als der **zum Eigenvektor v gehörende Eigenwert** bezeichnet wird.

Singulärwert-Zerlegung

(Singular Value Decomposition, SVD)

Die Matrix $Y_0 \in R^{nxN}$ kann **faktorisert** werden als $Y_0 = U\Sigma V^T$, wobei $U \in R^{nxn}$ und $V \in R^{NxN}$ **orthogonale Matrizen** sind und $\Sigma \in R^{nxN}$ abgesehen von der Diagonalwerten (den sogenannten **Singulär-Werten**) nur aus Nullen besteht.

Die Singulärwertzerlegung von Y_0 ist äquivalent zur **Eigenwertzerlegung** von $Y_0 T_0^T$.



Vergleich von PCA und DVAE

Lernen der Mannigfaltigkeit

PCA	DVAE
Kodierung/Dekodierung, keine Robustheit gegen Rauschen	nicht-linear, probabilistische Kodierung/Dekodierung mit Robustheit gegen Rauschen und nicht-linearen Aktivierungsfunktionen
unkorrelierte Koordinaten	korrelierte Ausgansdaten an der dünnsten Netzwerkebene
Koordinaten sind in absteigener Reihenfolge der Varianz geordnet	Koordinaten sind ungeordnet
die Spalten der Transformations-Matrix sind orthonormal	die Spalten der Transformations-Matrix sind nicht notwendigerweise orthonormal
Robustheit gegenüber moderatem Rauschen mit bekannten Verteilungen	Robustheit gegen eine Vielzahl verschiedener Arten und Größenordnungen an injeziertem Rauschen (masking noise, Gaussian noise, salt-and-pepper noise), da das Entrauschen entscheidung für die Generalisierung ist
einfacher Algorithmus (ohne Regularisierung), geringe Robustheit	die Punkte in niedrig-dimensionalen Mannifaltigkeiten sind robust gegen Rauschen im hoch-dimensionalen Beobachtungs-Raum

Vergleich zwischen PCA und DVAE

Training

PCA	DVAE
Abbildung der Eingangsdaten auf einen festen Vektor	Abbildung der Eingangsdaten auf eine Wahrscheinlichkeitsverteilung
iterative Methoden: QR Zerlegung, Jacobi Algorithmus, Singulärwertzerlegung	Fehlerrückführung (Backpropagation)
aufgrund der Kovarianz-Berechnung ineffizient bei großen Datenmengen	effizient bei großen Datenmengen aufgrund der starken Fähigkeit des Erlernens der Mannigfaltigkeit
basiert auf der Korrelations-/Kovarianz-Matrix, welche - zumindest in der Theorie - sehr empfindlich gegenüber Ausreißern sein kann	kann Beispiele direkt aus dem Eingangsraum generieren und daher die Eigenschfaten des Eingangsrauschens beschreiben ("reparametrization trick")