Deep Learning

Grundlagen Grundbegriffe Grundübungen

Alfatraining Kurs 2025-08 Dozent: Karsten Flügge

Themen des Tages

Tag 8

Transfer Learning
Finetuning: Anpassen von Modellen
(Un)überwachtes Vortrainieren

Auto-Encoder
Cluster
Explainable AI

Transfer Learning

Wiederverwendung eines vortrainierten Modells für eine neue Aufgabe

- Recycling **allgemeiner Merkmale** wie **Kanten**, Farben und Texturen
- die **Grundstruktur** des vorherigen Modells kann **beibehalten, gestaucht oder erweitert** werden

- Hilft **auch** bei **artfremd**en Daten!
  - z.B. **Audio ⇔ Text ⇔ Bild**

- Die **Gewichte** werden mit etwas **reduzierter Lernrate** trainiert
- **Reduziert die benötigte Datenmenge**
- **Reduziert die benötigte Rechenzeit**

Ablation

<u>Ablation:</u>
Herausfinden welche Architektur-Kompenenten oder Hyperparameter für gute/schlechte Ergebnisse verantwortlich sind.

Was hat den größten Einfluss?
Einzeln Änderungen 'ausschalten'/entfernen (ablating).

Ein bischen wie inverse Hyperparameter suche.

Transfer Learning

Feature Extraction:

- Verwendet das vortrainierte Modell als Feature-Extraktor.
- Die Gewichte der ursprünglichen **Schichten** werden **eingefroren.**
- **Nur die letzten Schichten** (oftmals Klassifikationsschichten) werden an die neue Aufgabe angepasst und **trainiert.**

Finetuning (Spezialfall des Transfer Learnings):

- Das **gesamte** vortrainierte **Modell** wird weiter **trainiert.**
- Alle oder einige der Gewichte des vortrainierten Modells werden weiter angepasst, um spezifische Merkmale der neuen Aufgabe zu lernen.

Low-rank adaptation (LoRA)

- spezielles Finetuning

Transfer Learning

- **Beispiel:**

Nutze allgemeinen Bild-Klassifizierer

"**lenet/resnet18**" **für** unsere **MNIST** Zahlen.

⚠️ resnet18 ist ein recht großes Netz, das training kann recht lange dauern (aber gute Resultate;)

Finetuning: Anpassen von Modellen

Finetuning trivial

Finetuning (Feinabstimmung) ist ein Prozess im maschinellen Lernen, bei dem ein bereits vortrainiertes Modell auf eine spezifische Aufgabe angepasst wird. Dies geschieht durch **weiteres Training des Modells** auf einer neuen, meist kleineren Datensatz. Der Vorteil liegt darin, dass das Modell bereits viele nützliche Merkmale gelernt hat und diese auf die neue Aufgabe übertragen kann.

Beispiele und Anwendungen

Bildklassifikation: Anpassung eines Modells, das auf ImageNet vortrainiert wurde, für spezifische Bildklassifikationsaufgaben.

Textverarbeitung: Finetuning von BERT oder **GPT** für spezifische NLP-Aufgaben wie Sentiment-Analyse oder Named Entity Recognition (NER).

⚠️ Finetuning von GPT für eigene Daten ist sehr zeitaufwändig!

⚠️ Finetuning bei großen **Foundation Models** meist nicht mehr sinnvoll, weil sie schon alles können.

Finetuning: Anpassen von Modellen

Arten des Finetunings

- **banal: einfach weiter trainieren mit checkpoint**
- **weiter trainieren alle Schichten**
- **weiter trainieren alle Schichten und zusätzliche**
- **weiter trainieren einige alte Schichten****
- **weiter trainieren nur neue Schichten****
- **weiter trainieren spezieller Schichten****
- **weiter trainieren 'Adapter' Pfade?**

****alte Schichten werden oft *eingefroren*.**

```
self.model.load_state_dict(torch.load(self.save_file, weights_only=True),strict=False)
```

Netzwerk Chirurgie

Netzwerk Chirurgie: modifizieren eines existierenden Modells.

Z.B. Schichten (vorn/hinten) anfügen, einfügen, erweitern

```
import torch
from torch import nn, optim

# z.B. ImageNet-Modell 1
old_model=torch.nn.Sequential(
  nn.Linear(128*128, 256),
  nn.Sigmoid(),
  nn.Linear(256, 100)
)

# fein-tuning für MNIST
new_model = torch.nn.Sequential(
  nn.Linear(28*28, 128*128), # hoch skalieren
  old_model, # altes ImageNet als Submodell
  nn.Linear(100, 10) # Klassen reduzieren
)
```

Netzwerk Chirurgie

Netzwerk Chirurgie: modifizieren eines existierenden Modells.

Z.B. Schichten (vorn/hinten) anfügen, einfügen, erweitern

```
import torch
from torch import nn, optim

# z.B. ImageNet-Modell 1
old_model=torch.nn.Sequential(
  nn.Linear(128*128, 256),
  nn.Sigmoid(),
  nn.Linear(256, 100)
)
```

manuell modifizieren

old_model['weights0']

Finetuning: Anpassen von Modellen

Neue Schichten können am Ende des Netzes angefügt werden oder innerhalb des Netzes,

Ein Spezialfall ist die **Verbreiterung von Linear Schichten**, so dass zu den ursprünglichen Gewichten neue Gewichte hinzugefügt werden welche dann trainiert werden:

$$W' = W + \mathbf{\Delta W}$$

ΔW ist unsere neue Gewichtsmatrix welche wir **feintunen.**
Die alten Gewichte W können wir '**einfrieren**'**,
also so ein einstellen dass die alten Gewichte beim Trainieren nicht mehr geändert werden.

** requires_grad = False

Netzwerk Chirurgie

Netzwerk Chirurgie: modifizieren eines existierenden Modells.

z.B. Schichten anfügen, einfügen, erweitern

```
import torch
from torch import nn, optim

# z.B. ImageNet-Modell 1
old_model=torch.nn.Sequential(
   nn.Linear(128*128, 256),
   nn.Sigmoid(),
   nn.Linear(256, 100)
)
```

manuell erweitern geht immer nach dem Laden des alten Modells:

```
layers = list(old_model.children())  # liste nun beliebig modifizierbar!

layers.insert(1, nn.Linear(256, 256))  # Neue Schicht zwischen bestehendem Modell

new_model=torch.nn.Sequential(layers)  # wieder Liste als Module
```

Netzwerk Chirurgie

Wir können eine alte Architektur in eine neue Laden wenn:
a) wir beim Laden strict=False setzen!

```
self.model.load_state_dict(torch.load(self.save_file), strict=False)
```

b) unsere Schichten Namen haben

```
model = nn.Sequential(OrderedDict([
  ("flat",nn.Flatten()),
  ("fc1",nn.Linear(28*28, 100)),
  ("relu1",nn.ReLU()),
  ("fc2",nn.Linear(100, 100)),
  ("relu2",nn.ReLU()),
  ("fc_extra",nn.Linear(100, 100)),
  ("relu_extra",nn.ReLU()),
  ("fc3",nn.Linear(100, 10))]))
)
```

sodass die passenden Gewichte beim Laden gefunden werden können

Finetuning: Anpassen von Modellen

FREEZE Technik:
Friere Originalschichten ein damit **nur neue Schichten trainiert** werden:

```
for param in original_layer.parameters():
  param.requires_grad = False

optimizer = optim.Adam(filter(lambda p: p.requires_grad, model.parameters()), lr=0.001)
```

Finetuning: Anpassen von Modellen

Learning Rate Technik:

```
model = nn.Sequential(
    nn.Conv2d(3, 32, 3),
    nn.ReLU(),
    nn.Linear(32 * 30 * 30, 100),
    nn.ReLU(),
    nn.Linear(100, 10),
)

optimizer = optim.Adam([
    {"params": model[0].parameters(), "lr": 1e-5},  # Conv2d
    {"params": model[2].parameters(), "lr": 1e-4},  # First Linear
    {"params": model[4].parameters(), "lr": 1e-3},  # Final Linear
])
```

"Vordere Schichten" sind zwischen Problemen ähnlich =>
diese nur langsam konservativ anpassen

SubModule

In pytorch kann man ein model einfach als Baustein für ein anderes verwenden!

```
sub_module=torch.nn.Sequential(
    nn.Linear(84, 84),
    nn.ReLU(),
    nn.Linear(84, 84),
    nn.ReLU()
)

# model adjusted for 3 channels and 32×32 input
model = torch.nn.Sequential(  # 32x32x3
    nn.Conv2d(3, 32, kernel_size=7, stride=1, padding="same"),
    nn.ReLU(),                # 32x32
    nn.MaxPool2d(2, 2),       # 16x16
    nn.Conv2d(32, 64, 3, 1, 1),
    nn.ReLU(),                # 16x16
    nn.MaxPool2d(2, 2),       # 8x8
    nn.Flatten(),             # 8*8*64 = 4096
    nn.Linear(64 * 8 * 8, 120),
    nn.ReLU(),
    nn.Linear(120, 84),
    nn.ReLU(),
     sub_module,
     sub_module,
    nn.ReLU(),
    nn.Linear(84, 10)         # CIFAR-10 has 10 classes
)
```

Aufgabe

a) Existierendes MNIST Modell Laden und dann **zwei Schichten einfügen** und weiter lernen.

z.b. mnist_cnn_freeze.py

```
torch.save(model, 'mnist_model_full.pt')
old_model = torch.load('mnist_model_full.pt')
```

b) Vorher alle Gewichte des alten Modells einfrieren:

```
for param in original_layer.parameters():
  param.requires_grad = False

optimizer = optim.Adam(filter(lambda p: p.requires_grad, model.parameters()), lr=0.001)
```

c) Extra Aufgabe:
FineTuning von existierendem resnet50 Modell, transfer für MNIST:

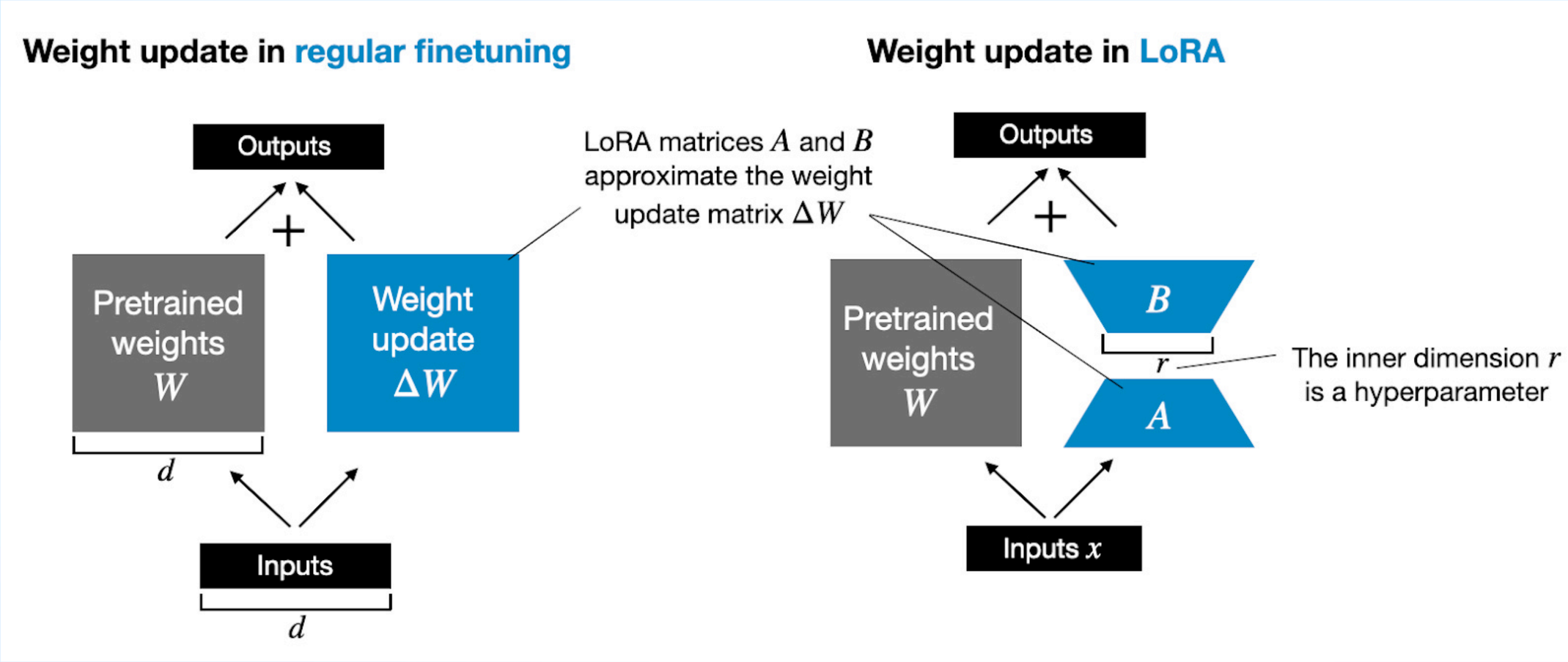
```
from torchvision.models import resnet50, ResNet50_Weights

old_model=resnet50(weights=ResNet50_Weights.IMAGENET1K_V1)
```

siehe transfer_learning.py für resnet18

Finetuning: LoRa

Low Rank Adoptation



Finetuning: LoRa

Vorteile:

- **Effizienz**: Reduziert die Anzahl der Parameter, die während des Fine-Tunings aktualisiert werden müssen.
- **Schnelleres Training &**
- **Speichereinsparung**: Nur die Low-Rank-Matrizen müssen gespeichert werden, nicht das gesamte Modell.
- **Modularität**: Die zusätzliche Struktur kann flexibel an unterschiedliche Modelle und Aufgaben angepasst werden.
- **Geringerer Overhead**: Durch die Einführung einer geringeren Anzahl von Parametern wird das Risiko von Overfitting reduziert.

Nachteile:

- **Modellkomplexität**: Die Einführung von Low-Rank-Matrizen kann die Modellstruktur komplexer machen, was das Debugging erschwert.
- **Möglicher Leistungsverlust**: Da nicht alle Parameter des Modells angepasst werden, kann in bestimmten Fällen die Feinabstimmung weniger effektiv sein. Underfitting.

Finetuning: LoRa

Anstatt alle Gewichte eines vortrainierten Modells zu aktualisieren, fügt **LoRA** spezialisierte **kleine Matrizen (Low-Rank Matrizen)** in das Modell ein.

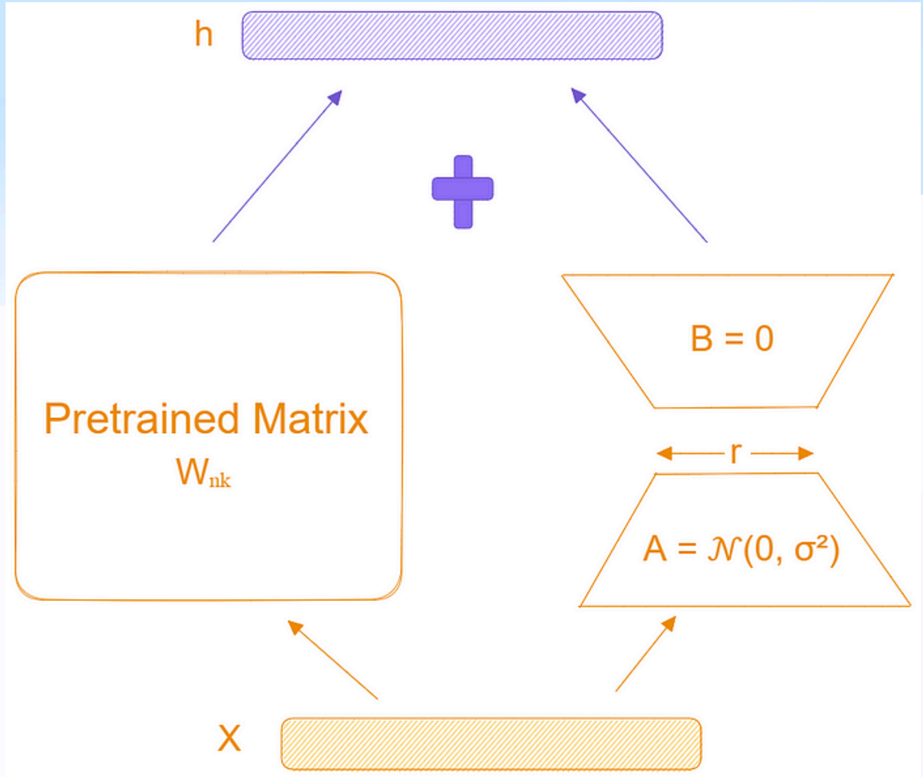
So kann beispielsweise eine gigantische Matrix W_0 von 2048 * 2048 Gewichten aufgespalten werden in **zwei lernbare Matrizen A und B:**

$$W' = W_0 + A*B$$

wobei die Dimension von A und B sehr klein ist:
A : **2048 * r** und
B : **r * 2048** (z.B. **rank** r = 4)

Diese werden initialisiert mit Gewichten
A : N(0,σ²)
B : 0

D.h. am Anfang fällt der Zusatzterm A*B weg
(es fand noch kein Finetuning statt)



Finetuning: LoRa

W' = W + **ΔW**

W' = W_0 + **A*B** **mathematischer Hintergrund:**

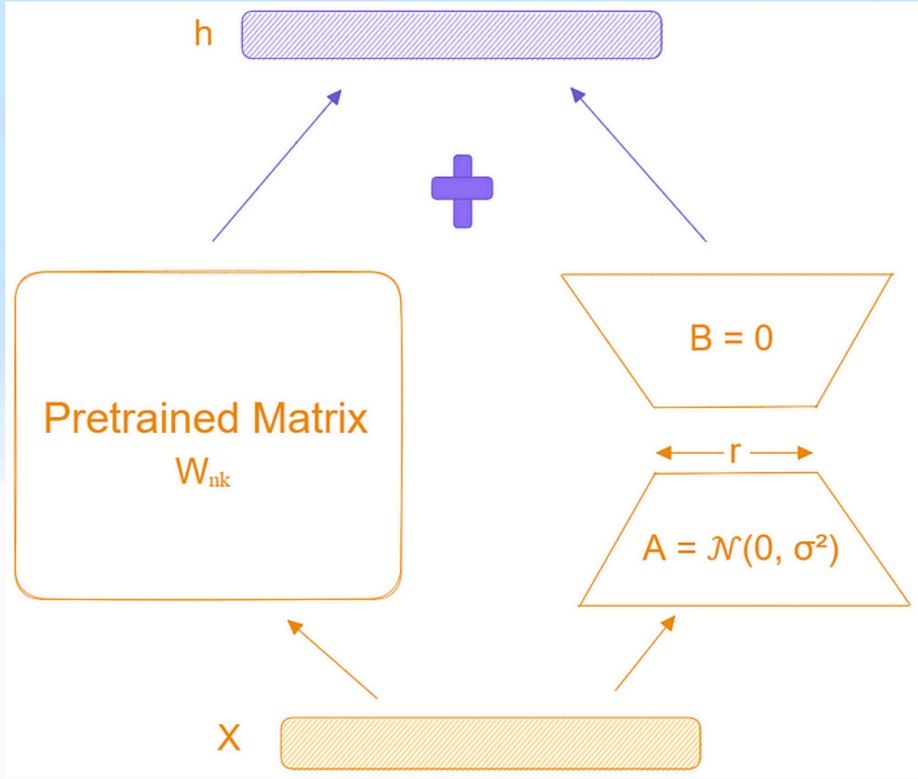
ΔW : n*m Matrix mit n,m riesig
A : n*r Matrix r kleiner 'rank'
B : r*m Matrix
A * B hat Dimension **n*m** (@ in numpy)
(* normale Matrix Multiplikation,
nicht Element-Weise)

A : 3*1 [1,2,3]
B : 1*3 (2,3,4)

B*A = 2+6+12

A*B = [(2,3,4)
(4,6,8)
(6,9,12)] 3*3 Matrix

Große Matrix wird aus relativ kleinen Matrizen "**aufgespannt**"



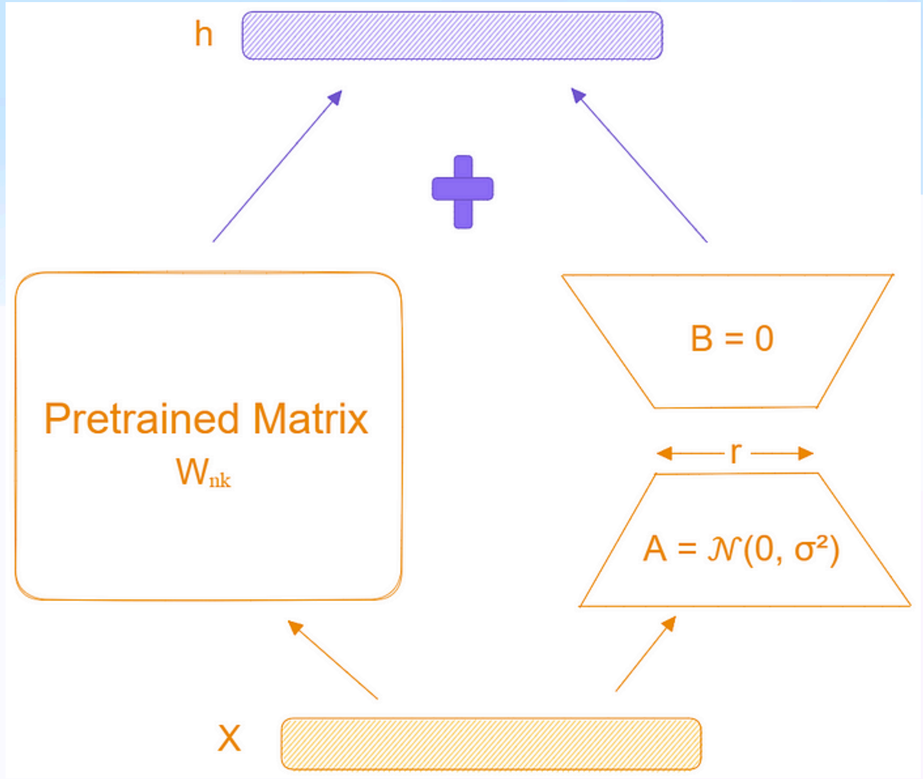
Pruning

Die selbe Idee von **LoRA** wird auch angewendet um Netze zu **schrumpfen** von Netzen (z.B. in Attention Blocks)

W' = A*B

wobei die Dimension von A und B sehr klein ist:
A : **2048 * r** und
B : **r * 2048** (z.B. **rank** r = 4)

Dann verliert man etwas Kapazität,
aber es die resultierende Matrix ist
oft 'mächtig genug'



Finetuning: LoRa

```python
# Implementierung von LoRA:
class LoRALayer(torch.nn.Module):
  def __init__(self, original_layer, rank):
    super(LoRALayer, self).__init__()
    self.original_layer = original_layer
    for param in original_layer.parameters():
      param.requires_grad = False
    # ^^ Freeze der originalen Layer:
    # nur die LoRA-Matrizen werden trainiert

    # Low-Rank Matrizen
    self.lora_A = torch.nn.Parameter(torch.randn((original_layer.in_features, rank)))
    self.lora_B = torch.nn.Parameter(torch.randn((rank, original_layer.out_features)))

  def forward(self, x):
    # Original Layer wird unverändert verwendet
    original_output = self.original_layer(x)
    # Hinzufügen der Low-Rank-Adaptation
    lora_output = torch.matmul(torch.matmul(x, self.lora_A), self.lora_B)
    return original_output + lora_output
```

Finetuning: LoRa

LoRa in der Praxis:

https://arxiv.org/abs/2106.09685
Das Paper ist von 17 Jun ***2021***,
rasante Erfolgsgeschichte!

https://github.com/microsoft/**LoRA**

pip install **loralib**

Aktuell ist LoRa extrem beliebt zun finetunen von Sprachmodellen (LLMs).

Finetuning: LoRa

Eine schöne alternative community Bibliothek welche das Wrappen von Schichten mit Laura automatisch bewerkstelligt:

https://github.com/fkodom/lora-pytorch

Schaffen wir das selber?

Aufgabe: LoRa

Wrappen von Schichten eines vortrainierten Modells mit LoRa … + Training ("fine tuning")

(Un)supervised Learning

Überwachtes Lernen (Supervised Learning)

- **Label / Kategorien vorhanden**
- **gekennzeichneten Trainingsdaten**

Unüberwachtes Lernen (Unsupervised Learning)

- **Nur Rohdaten Vorhanden**
- **<u>nicht</u> gekennzeichnete Trainingsdaten**

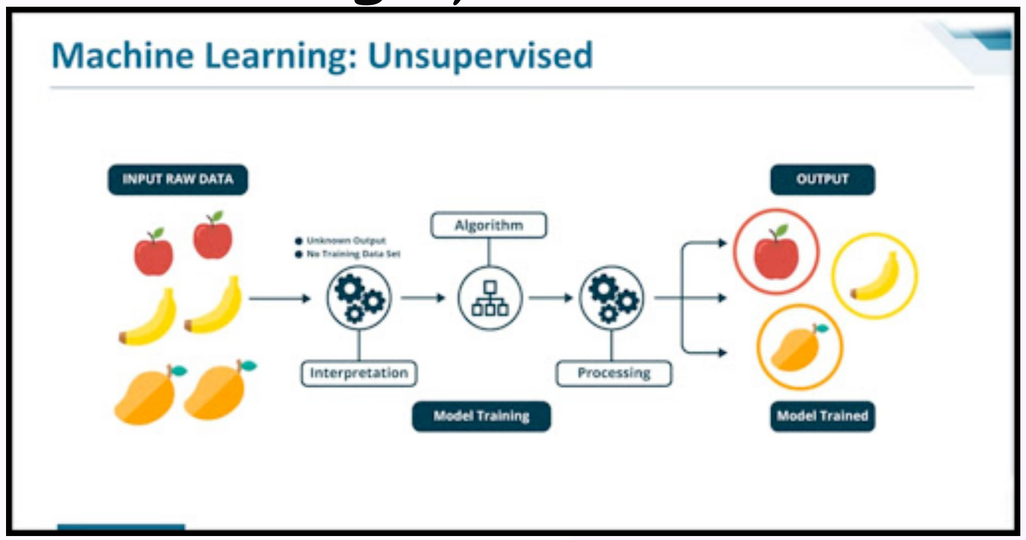
… semi-supervised learning

… selbstüberwachtes Lernen (self-supervised learning)

Unsupervised Learning

Anwendugen Unüberwachtes Lernen

- **Ähnlichkeits Erkennung**
- **Cluster Erkennung ≈ PCA …**
- **Label** können nachträglich **manuell** gegeben werden
- **Daten Bereinigung:** Ausreisser finden …
- **Vortrainieren** (z.B. bei Transfer Learning)
- **Autoencoder** (Kompression / **Embeddings**)



Unüberwachtes Vortrainieren

Unüberwachtes Vortrainieren ist eine Methode im Deep Learning, bei der neuronale Netzwerke auf **unbeschrifteten Daten** vortrainiert werden, bevor sie auf eine **spezifische, beschriftete Aufgabe feinabgestimmt** werden. Diese Technik kann helfen, die Lernleistung zu verbessern, insbesondere wenn die Menge an beschrifteten Daten begrenzt ist.

Unüberwachtes Vortrainieren

Explainable AI

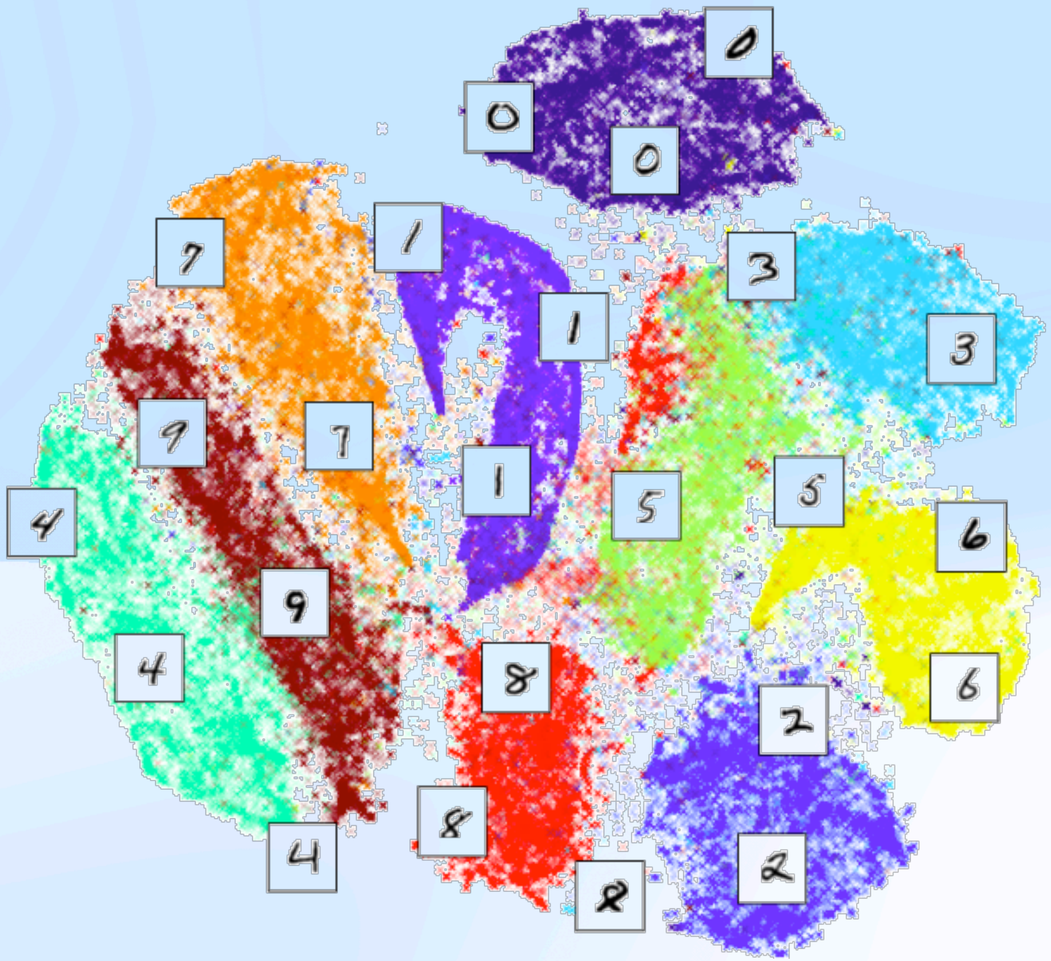
Clustering

Clustering ist ein **Unsupervised Learning** Technik, dass heisst man kann **aus Rohdaten** ohne Zusatzdaten (Label, …) **Erkenntnisse** Gewinnen.

Ähnliche Daten werden **automatisch** zu Clustern zusammengefasst!

⚠️💡 Meist muss die **Anzahl der Cluster** mitgegeben werden, da ad hoc nicht klar ist in welcher Auflösung geclustert werden soll

Clustering



Nachträgliches manuelles Labeln

Clustering

Automatisch Anzahl der Cluster Bestimmen

1. Elbow-Methode (Ellbogen-Methode)

- **Idee:** Diese Methode versucht, die optimale Anzahl von Clustern zu finden, indem der **Wert der Trägheit** oder der **Summe der quadratischen Abstände** zwischen den Punkten und ihren zugehörigen Clusterzentroiden für verschiedene Anzahlen von Clustern berechnet wird. Man wählt dann die Anzahl von Clustern, bei der eine deutliche Verringerung der Kosten auftritt (der “Ellbogenpunkt” im Plot).
- **Vorteile:** Einfach umzusetzen und zu interpretieren.
- **Nachteile:** Kann subjektiv sein, da der Ellbogenpunkt nicht immer klar definiert ist.

2. Silhouettenanalyse

- **Idee:** Die Silhouettenanalyse bewertet, wie gut jedes Datenobjekt innerhalb seines Clusters liegt, indem die durchschnittliche Entfernung zwischen den Punkten im gleichen Cluster und den Punkten im nächstgelegenen Cluster berechnet wird. Der **Durchschnitt aller Silhouettenkoeffizienten** für verschiedene Clusteranzahlen kann geplottet werden, um die beste Clusteranzahl auszuwählen.
- **Vorteile:** Liefert eine numerische Metrik, die die Qualität der Clusterbildung direkt misst.
- **Nachteile:** Kann für große Datensätze rechenintensiv sein.

3. Gap-Statistik

- **Idee:** Die Gap-Statistik vergleicht die Trägheit eines gegebenen Clusters im Vergleich zu einem zufälligen Referenzcluster. Die Anzahl der Cluster wird gewählt, bei der der Gap-Wert maximiert wird, d. h. die größte Abweichung von einer zufälligen Clusterstruktur.
- **Vorteile:** Statistisch fundierter als die Elbow-Methode.
- **Nachteile:** Rechenaufwändiger, da viele Referenzdatensätze generiert und verglichen werden müssen.

4. Hierarchisches Clustering mit Dendrogramm

- **Idee:** Beim hierarchischen Clustering wird ein Dendrogramm erstellt, das die Hierarchie von Clustern zeigt. Die optimale Anzahl der Cluster kann durch das Schneiden des Dendrogramms an einem bestimmten Punkt (z. B. basierend auf einer Distanzschwelle) ermittelt werden.
- **Vorteile:** Keine direkte Angabe der Clusteranzahl erforderlich.
- **Nachteile:** Schwer skalierbar für sehr große Datensätze.

5. DBSCAN (Density-Based Spatial Clustering of Applications with Noise)

- **Idee:** DBSCAN ist ein **dichtebasiertes Verfahren**, das keine Angabe der Anzahl der Cluster erfordert. Stattdessen werden zwei Parameter benötigt: die minimale Anzahl von Punkten, die einen Cluster bilden, und ein Epsilon-Wert, der den maximalen Abstand zwischen zwei Punkten bestimmt, damit sie als Teil des gleichen Clusters betrachtet werden.
- **Vorteile:** Automatische Clusteranzahlbestimmung und Identifizierung von Ausreißern.
- **Nachteile:** Sensibel für die Wahl der Parameter.

6. BIC/AIC (Bayesian Information Criterion/Akaike Information Criterion)

- **Idee:** Diese Ansätze verwenden probabilistische Modelle wie das **Gaussian Mixture Model (GMM)**. Die optimale Anzahl der Cluster wird durch Minimierung des BIC- oder AIC-Werts bestimmt, der die Modellkomplexität und die Anpassungsgüte des Modells berücksichtigt.
- **Vorteile:** Statistisch fundierte Auswahl der Anzahl von Clustern.
- **Nachteile:** Rechenintensiv, insbesondere für große Datensätze.

Automatisch Anzahl der Cluster Bestimmen

1. Elbow-Methode (Ellbogen-Methode)

- **Idee:** Diese Methode versucht, die optimale Anzahl von Clustern zu finden, indem der **Wert der Trägheit** oder der **Summe der quadratischen Abstände** zwischen den Punkten und ihren zugehörigen Clusterzentroiden für verschiedene Anzahlen von Clustern berechnet wird. Man wählt dann die Anzahl von Clustern, bei der eine deutliche Verringerung der Kosten auftritt (der “Ellbogenpunkt” im Plot).
- **Vorteile:** Einfach umzusetzen und zu interpretieren.
- **Nachteile:** Kann subjektiv sein, da der Ellbogenpunkt nicht immer klar definiert ist.

Clustering

Anwendungen

- Daten **Verständnis**
- Kunden**segmentierung** im Marketing
- (Bild**kompression)**
- **Anomalie**erkennung (Ausreisser / Konfusion)
- Muster**vorhersage** in Finanzdaten
-

Clustering

Density-based
Distribution-based
Centroid-based

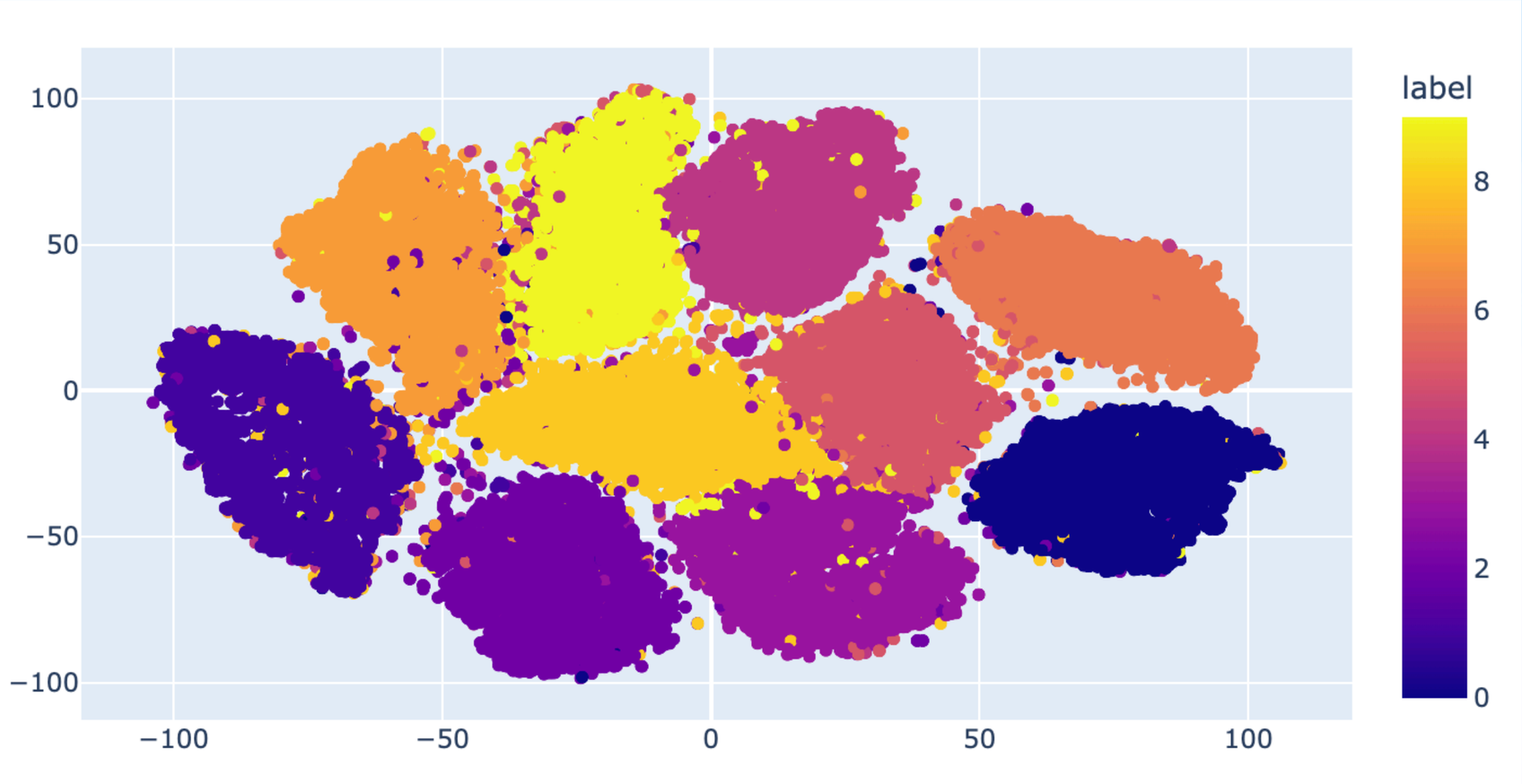
PCA
t-SNE
Autoencoder
K-Means

```
Birch
AffinityPropagation
DBSCAN
OPTICS
MeanShift
AgglomerativeClustering
```

Clustering

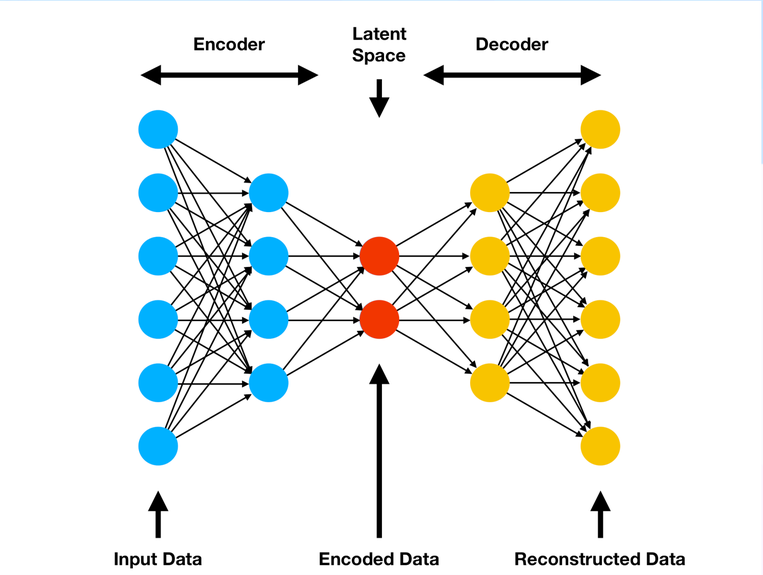
Good to find **confusing examples** (1 ≈ 7)

Good to find **bad labels** (human errors)



Autoencoder

Ein Autoencoder ist ein neuronales Netzwerk, das darauf trainiert ist, Daten durch **Kompression** und anschließende **Rekonstruktion** (Identität) zu lernen.



Bestandteile:
Encoder
Embedding / Latent Space
Decoder

Ziel: Minimierung des Rekonstruktionsfehlers

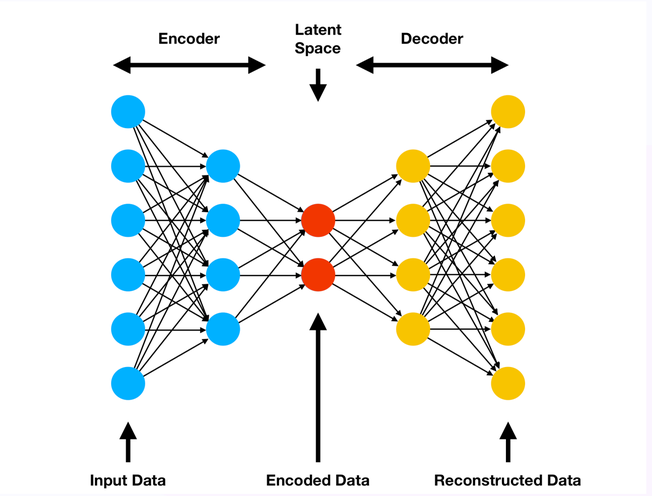
Embedding / Latent Space

Im **Embedding / Latent** Space liegt die **Kondensierte Information.**

Abstrakter Raum. Kann für verschiedene Modi genutzt / gekoppelt werden.

Morphing: zwischen zwei **Embeddings interpolieren**

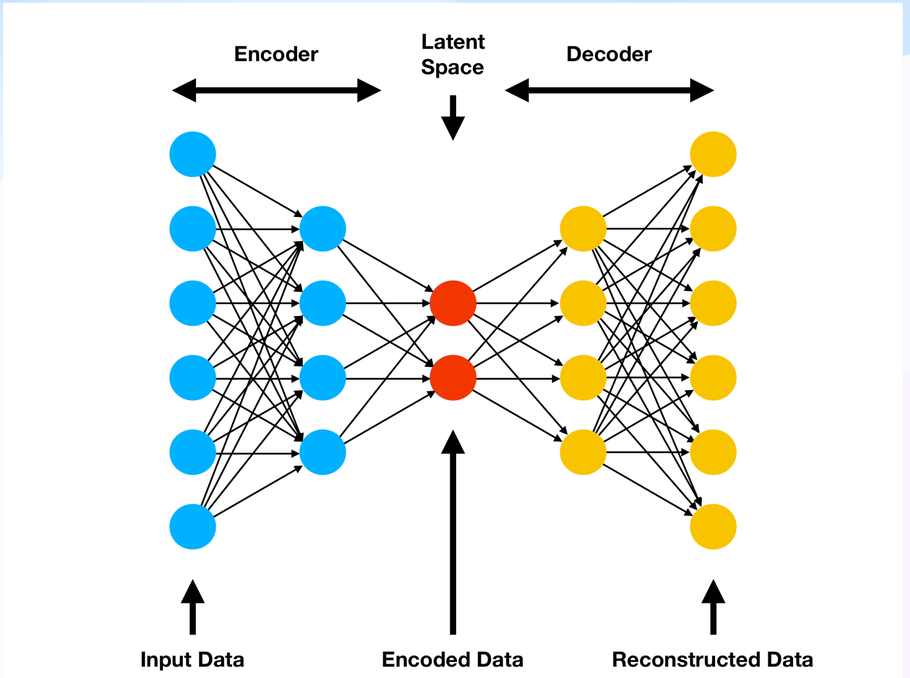
Mit Decoder:
Möglichkeitsraum für Rekonstruktionen.



Embedding / Latent Space

Morphing: zwischen zwei **Embeddings interpoliere**

```
c = Encoder(cat)
g = Encoder(giraffe)
Decoder((c+g)/2)
```



Embedding / Latent Space

Morphing: zwischen zwei **Embeddings interpoliere**

```
two = Encoder("2")
seven = Encoder("7")
mixed = Decoder((two + seven)/2)
```

Autoencoder

Anwendungen

- Datenkompression:
  Verlustbehaftete Kompression
  ⚠️ Achtung man kann von kleinen Fehlern ausgehen!
  ⚠️ nicht für .exe .pdf … zip wo jedes Byte zählt
- Reduktion der Datenkomplexität
- Merkmalsextraktion (Clustering)
- Ähnlichkeitssuche (RAG Embeddings...)
- Rauschreduktion (Denoising, Weichzeichnen)
- Multimodales Embedding (Bild≈Text!)
- Generative Modelle (Text zu Bild)

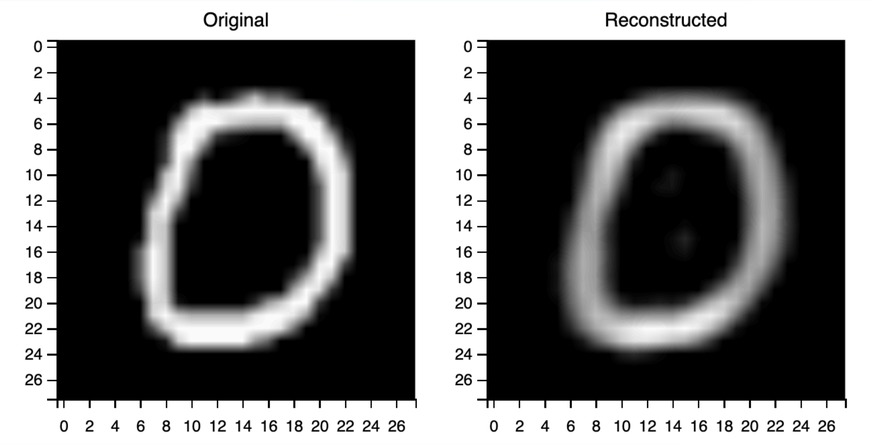
Autoencoder

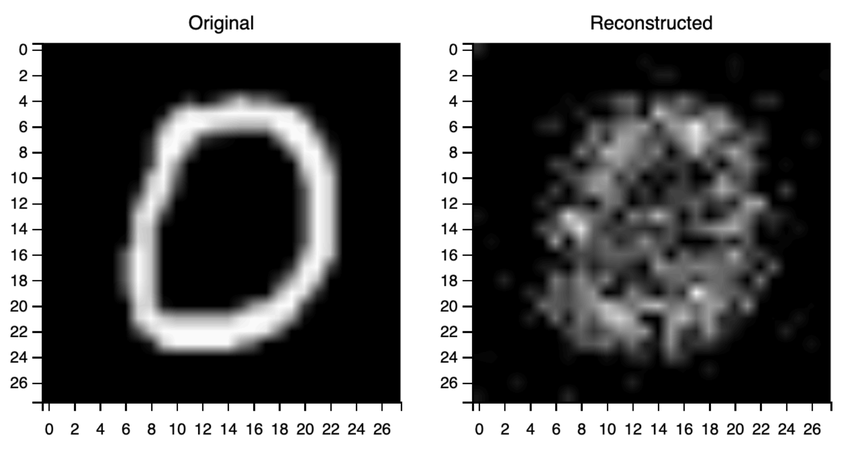
Nachteile:

Kann anfällig für **Überanpassung** sein.

=> Braucht Regularisierung!

Die **Wahl** der **Dimensionalität** des latenten Raums ist oft nicht trivial.

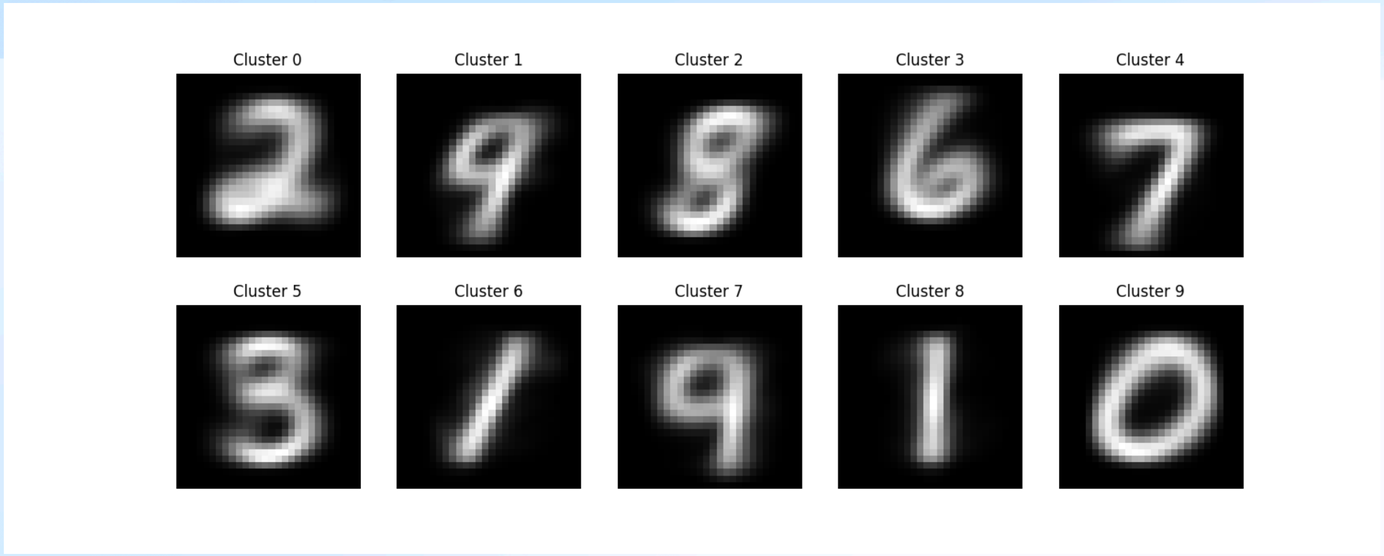




Autoencoder

Extremfall MNIST:
Bild -> Zahl 0…9 -> Bild

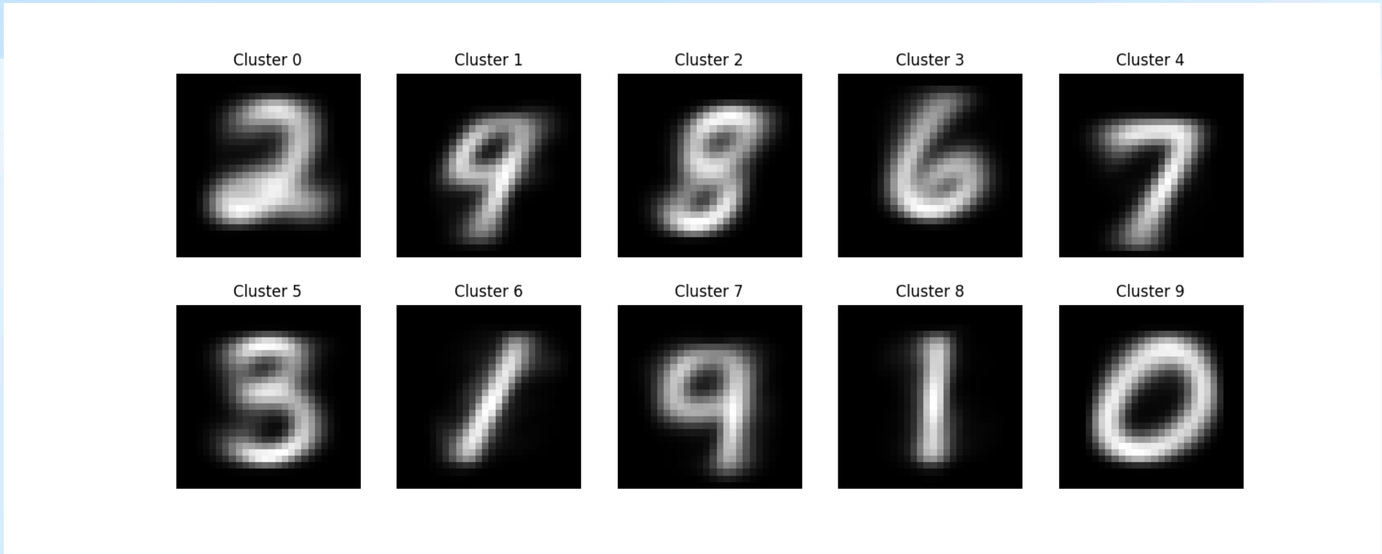
Exakte Rekonstruktion natürlich idR nicht mehr möglich

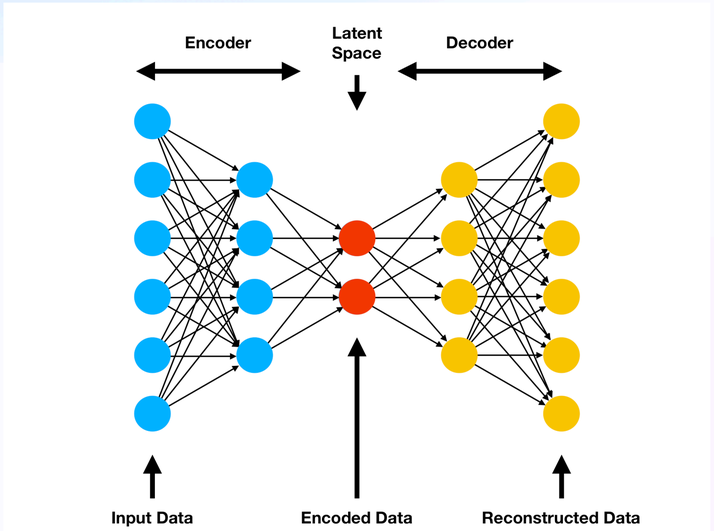


Autoencoder

Realistisch MNIST:
Bild -> Zahl 0…9 **One-Hot** + '**style**' -> Bild

Idealerweise kann die interne Representation jedes Bildes als **Zahl** + **Stil** embedded werden

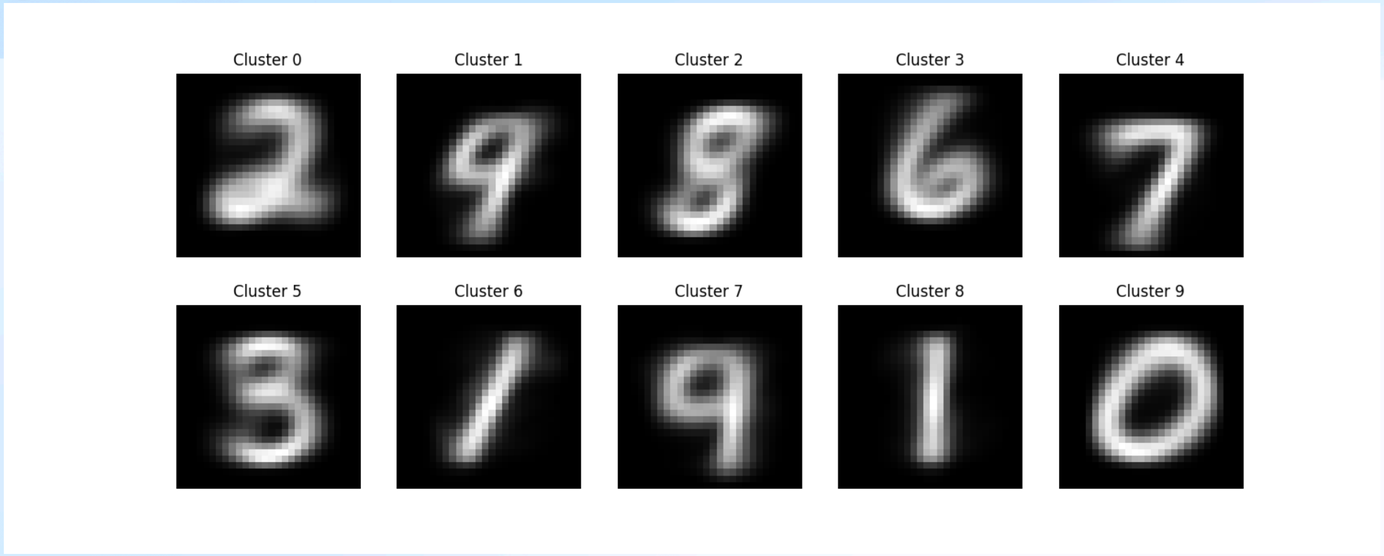


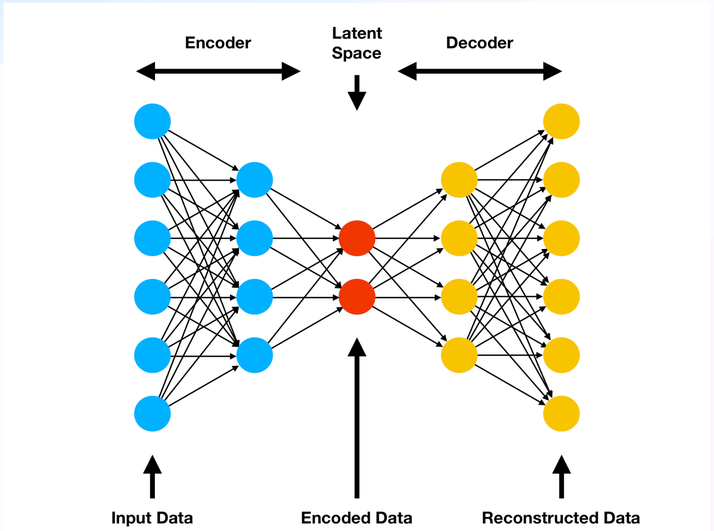


Autoencoder

Typisches Encoding:
Bild -> **Dense** embedding -> Bild

Idealerweise kann die interne Representation jedes Bildes als Zahl + Stil embedded werden





Autoencoder

One-Hot embedding:
Verteile n Klassen in Vektor der länge n.
Ein Eintrag ist dann
1 für "ja, klasse aktiv"
0 für "nein, klasse inaktiv"
Die Position der eins steht für die Klasse:

- Cat -> [1, 0, 0]
- Dog -> [0, 1, 0]
- Fish -> [0, 0, 1]

One-Hot embedding ist der output von Softmax

One-Hot embedding als Input z.B. Ascii Code
Vektor der Länge 128 pro Buchstabe

Klingt etwas verschwenderisch, aber Netz kann deutlich besser damit rechnen.

Siehe sparse matrices (in Graph Neural Networks (GNNs), Sparse Attention…)

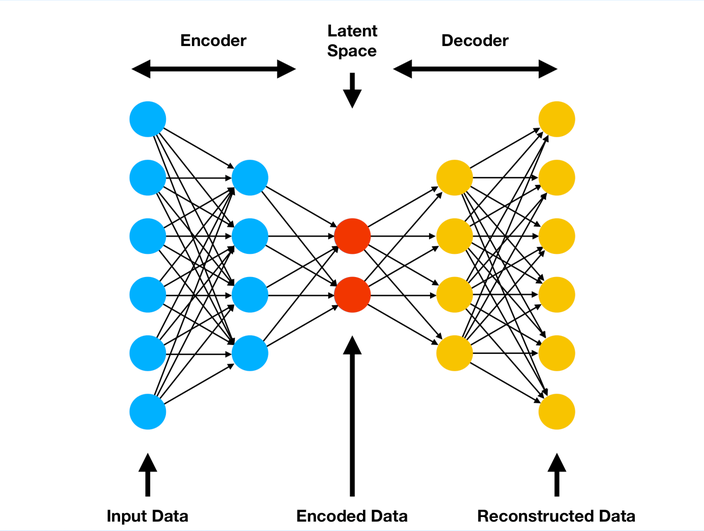
Autoencoder

Satz von Idealer Kompression:

"Autoencoder neigen dazu zur *idealen* Representation zu konvergieren"

Ideal im Sinne von Informationstheorie ("Huffman Code")

Autoencoder Fehlerfunktion

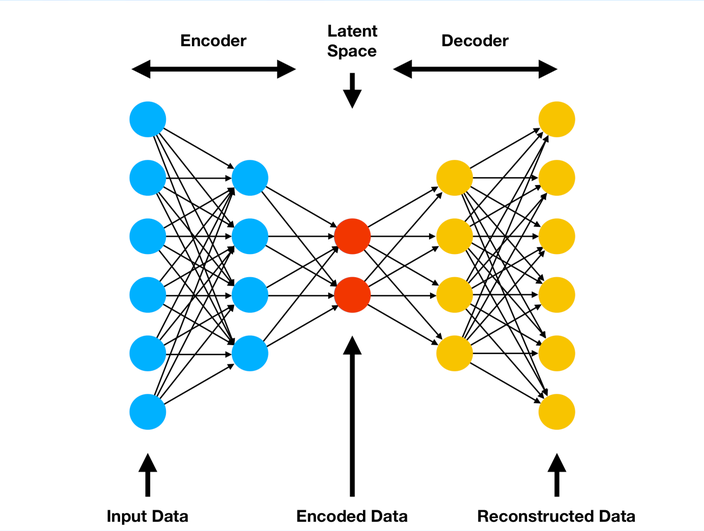


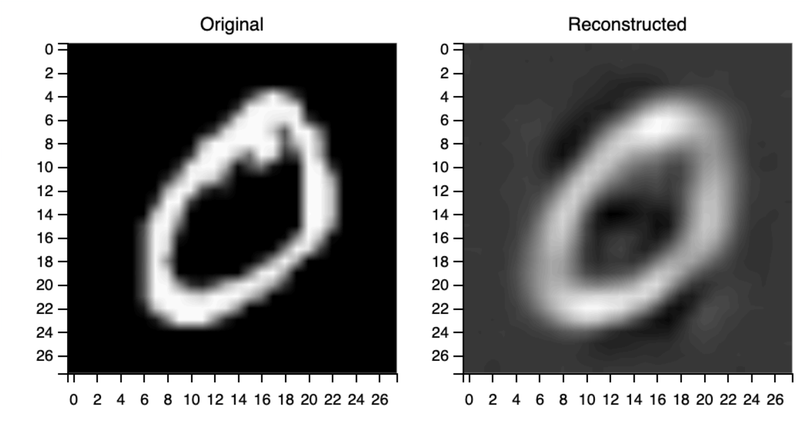
Mittlere Schicht "Innere Repräsentation"

Naiver Ansatz: Punktweise Differenz / MSE
Später: Custom Criterion
(Translationsinvarianz etc etc … …)

Aufgabe

Trivial MNIST Autoencoder 28*28 -> 20 -> 28*28





Naiver Ansatz Fehler: Punktweise Differenz / MSE

Latent Space: Suche schöne Beispiele

Achtung Tipp: Y Target ist wieder X selbst!

Ziel: fallender Fehler (erst mal abstrakt)

Explainable AI

Explainable AI

Erkläre was unsere Netze tun

vs

Blackbox

Explainable AI

Visualisiere welche Gewichte im trivial MNIST für die Klassifizierung relevant waren.
Aktivierung des Layers darstellen.

Explainable AI

Explainable AI (XAI), oder erklärbare Künstliche Intelligenz, bezieht sich auf Techniken und Methoden, die es ermöglichen, die **Entscheidungen und das Verhalten** von KI-Modellen **nachvollziehbar** und verständlich zu machen. XAI ist besonders wichtig in Bereichen, in denen **Vertrauen**, **Transparenz** und ethische Überlegungen eine große Rolle spielen, wie zum Beispiel im Gesundheitswesen, in der Justiz oder im Finanzwesen.

Explainable AI

- Keine universell anerkannte Definition
- Keine rechtlich bindenden Regeln
-

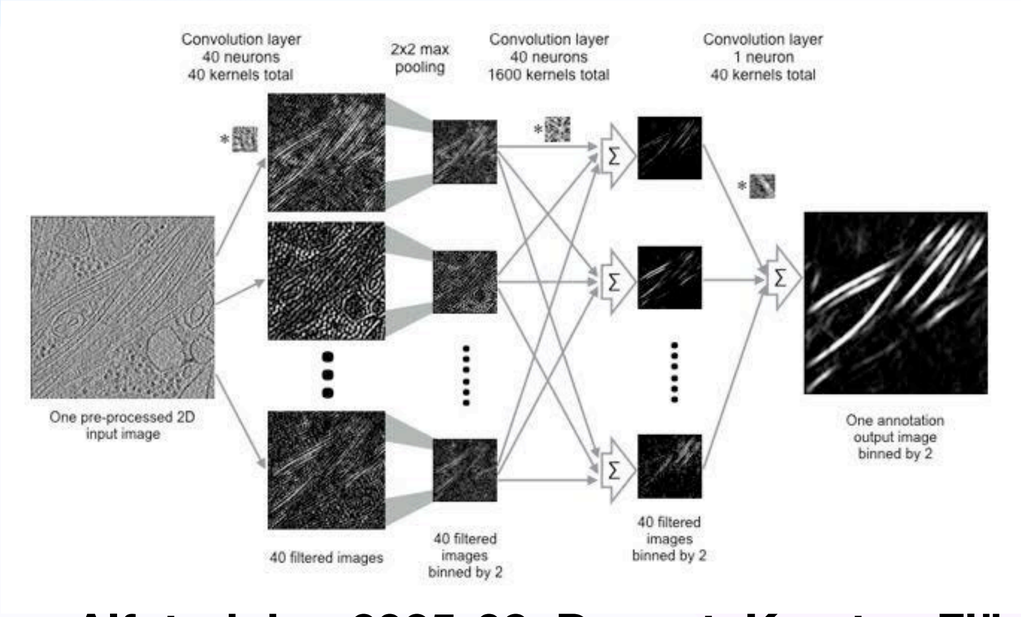
Explainable AI

Methoden der Erklärbarkeit:

- **Modelle mit intrinsischer Erklärbarkeit:** Diese Modelle sind von Natur aus einfacher zu verstehen, z.B. Entscheidungsbäume, lineare Modelle.

- **Post-hoc-Erklärungen:** Diese Techniken erklären die Entscheidungen von komplexen Modellen (wie tiefen neuronalen Netzen) nachträglich. Dazu gehören Methoden wie
  - **LIME** (Local Interpretable Model-agnostic Explanations) und
  - **SHAP** (SHapley Additive exPlanations).

- **Nachvollziehen** der Berechnungen
  - Einfluss (**impact**) Analyse
  - Lokalisierung
  - "Grossmutter-Neuron"
  - Einfluss Rückverfolgung
  - **CNN Schichten!**



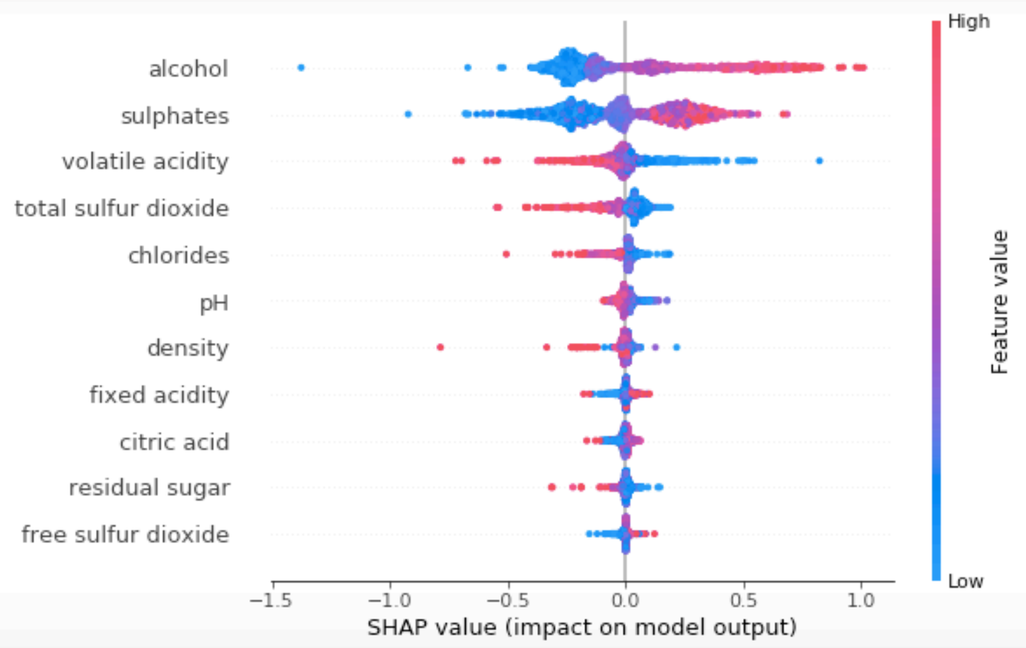
Explainable AI

Jede Form von **Visualisierung** kann helfen!

Latent Space **Visualisierung**

Cluster!

K-Means Clustering



Explainable AI

Vorteile:

- Erhöhtes **Vertrauen** und Akzeptanz
- Bessere **Fehleranalyse** und Debugging
- Erfüllung **regulatorischer** Anforderungen

Nachteile:

- **Komplexität** der Implementierung
- Mögliche **Einschränkungen** bei der Modellleistung
- z.B. keine Fully Connected layers mehr?

Explainable AI

Beispiele und Anwendungen:

- Gesundheitswesen: Diagnoseerklärungen
- Finanzwesen: Kreditwürdigkeitsprüfungen
- Justiz: Strafmaßentscheidungen
- Versicherungen:

Explainable AI

Beispiel MNIST:

```
import shap
import cv2 # ⚠️ pip install opencv-python !!!
```

Explainable AI

Neuste Entwicklung:

Sprachmodelle erklären sich selbst.

A New Type of Neural Network Is More Interpretable + Diskussion