

1 Preface

1.1 Formeln

$$\text{Euklidische Norm} = \sum_{l=1}^n |x_l^i - x_l^j| \quad (1)$$

$$\text{Euklidische Distanz} = \|\cdot\| = \sqrt{(x_1^i - x_1^j)^2 + \dots + (x_n^i - x_n^j)^2} \quad (2)$$

$$\text{z-Transformation} = \hat{x}_j^{(i)} = \frac{x_j^{(i)} - \bar{x}_j}{\sigma_j} \quad (3)$$

1.2 Standardimporte

2 Supervised Learning

Ensemble-Lernen: viele verschiedene Modelle werden gleichzeitig gelernt und die Resultate bei der Anwendung geeignet aggregiert. Es kann insbesondere auch dazu genutzt werden, der Überanpassung einzelner Modelle entgegenzuwirken. Dazu werden die verschiedenen Modelle nur mit einer Teilmenge der zur Verfügung stehenden Daten trainiert (Ziehen mit Zurücklegen \rightarrow *bootstrap aggregating* oder *bagging*).

2.1 Lineare/Polynomiale Regression

Optimale Anpassung einer Geraden an eine gegebene Menge an Punkten, d.h. für eine Funktion

$$h_{\Theta}(x) = \Theta_0 + \Theta_1 x_1 + \dots + \Theta_n x_n$$

soll der Parametervektor Θ gefunden werden (mit Θ_0 als Konstante), der die Summe der quadrierten Abweichungen der Funktionswerte $h_{\Theta}(x)$ von den tatsächlichen Werten y minimiert (Methode der kleinsten Quadrate):

$$\min_{\Theta} L(D, f) = \min_{\Theta} \sum_{i=1}^m (f(x^{(i)}) - y^{(i)})^2$$
$$\min_{\Theta} L(D, \Theta) = \min_{\Theta} \|X_D \Theta - y_D\|^2$$

wobei X_D eine Matrix mit den Eingabedaten (zzgl. führende 1-Spalte) und y_D der Vektor der tatsächlichen Werte ist:

$$X_D = \begin{pmatrix} 1 & x_1^{(1)} & \dots & x_n^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & \dots & x_n^{(m)} \end{pmatrix}, \quad y_D = \begin{pmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{pmatrix}$$

Lokales Minimum = Globales Minimum, da die Kostenfunktion konvex ist. Lösung numerisch oder per *Gradient Descent* \rightarrow ist bei großen Trainingsdatensätzen und/oder vielen Attributen die praktikabelste Methode (s. Skript S. 13: $\nabla_{\Theta} L(D, \Theta) = 0 \Leftrightarrow (X_D^T X_D)^{-1} X_D^T y_D = \Theta$, wobei inverse von $X_D^T X_D$ sehr rechenaufwändig ist).

Erweiterung auf Polynome höheren Grades durch (Kreuz-)Multiplikation bestehender Merkmale \rightarrow Modell ist linear bzgl. des erweiterten Merkmalsraums und erscheint polynominal bei Projektion auf den ursprünglichen Merkmalsraum.

Evaluation mittels **Bestimmtheitsmaß** (=normalisierte Variante des quadratischen Fehlers):

$$R^2(D, f) = 1 - \frac{\sum_{i=1}^m (f(x^{(i)}) - y^{(i)})^2}{\sum_{i=1}^m (y^{(i)} - \bar{y})^2}$$

mit $\bar{y} = \frac{1}{m} \sum_{i=1}^m y^{(i)}$, wobei in der Praxis der Durchschnitt mehrerer R^2 berechnet wird (*Kreuzvalidierung*).

- $R^2(D, f)$ ist maximal 1 $\rightarrow f$ modelliert D perfekt
- $R^2(D, f) = 0 \rightarrow$ naives Modell, f sagt stets den Mittelwert \bar{y} voraus
- $R^2(D, f) < 0 \rightarrow$ Modell schlechter als naives Modell
- $R^2(D^{\text{train}}, f)$ sollte relativ nahe an 1 liegen
- $R^2(D^{\text{test}}, f)$ ist üblicherweise kleiner als $R^2(D^{\text{train}}, f)$
- Je näher $R^2(D^{\text{test}}, f)$ an $R^2(D^{\text{train}}, f)$, desto besser ist das Modell generalisiert

Überanpassung: Modell passt sich zu stark an Trainingsdaten an, d.h. es wird zu komplex modelliert. Dies führt zu schlechterer Generalisierung auf Testdaten \rightarrow *Varianzfehler*

Unteranpassung: Modell ist nicht ausdrucksstark genug; Trainings- und Testdaten werden unzureichend modelliert \rightarrow *Verzerrungsfehler*

Ermittlung der **optimalen Modellkomplexität** durch Betrachtung der Kostenfunktionswerte oder Bestimmtheitsmaße bei steigender Komplexität:

- Trainingsdaten: Je komplexer das Modell, desto höher die Bestimmtheit
- Testdaten
 - Bestimmtheit nimmt zunächst ebenfalls zu (das Modell ist noch unterangepasst)
 - Ab einem gewissen Punkt nimmt die Bestimmtheit ab: Das Modell ist überangepasst
- Optimaler Punkt: Modellkomplexität, bei der die Bestimmtheit bzgl. der Testdaten maximal ist

Automatische Lösung des *Verzerrungs-Varianz-Dilemmas* durch **Regularisierung**: Hinzufügen eines mit λ (*Regularisierungsparameter*) gewichteten Strafterms (*Tikhonov-Regularisierer* R_T) zur Kostenfunktion, der die Größe der Parametervektoren begrenzt. Sog. *Ridge-Regression*:

$$L_T(D, \Theta) = \|X_D \Theta - y_D\|^2 + \lambda \sum_{i=1}^n \Theta_i^2$$

- Regularisierer wird ohne Θ_0 berechnet
- Je mehr $\Theta_i \neq 0$, desto größer wird der Tikhonov-Regularisierer \rightarrow Kosten steigend
- Einbeziehung von $\lambda \sum_{i=1}^n \Theta_i^2$ erzwingt Fokussierung auf möglichst einfache Funktionen
- Kleines $\lambda \rightarrow$ Überanpassung, großes $\lambda \rightarrow$ Unteranpassung

2.2 Logistische Regression

Diskreter Wertebereich der Zielvariablen $y^{(i)}$, idR. endlich und oft auch nur binär ($y^{(i)} \in \{0, 1\}$). Klassen haben idR. keine (eindeutige) Ordnung.

Einsetzen eines linearen Modells h_Θ in eine Funktion $g(z) = \frac{1}{1+e^{-z}}$ mit dem Zielbereich $(0, 1)$ ergibt sog. *Sigmoid-Funktion*:

$$h_\Theta^{\text{logit}}(x) = \frac{1}{1 + e^{-(\Theta_0 + \Theta_1 x_1 + \dots + \Theta_n x_n)}}$$

Klassifikation durch Schwellwert 0.5:

$$\text{clf}_f(x) = \begin{cases} 1 & \text{falls } h_\Theta^{\text{logit}}(x) \geq 0.5 \\ 0 & \text{falls } h_\Theta^{\text{logit}}(x) < 0.5 \end{cases}$$

bzw. bei n Klassen diejenige Klasse k , für die $h_\Theta^{\text{logit}}(x)_k$ maximal ist. Bewertung mittels der *logistischen Kostenfunktion* L^{logit} :

$$L^{\text{logit}}(D, f) = - \sum_{i=1}^m \left[\underbrace{y^{(i)} \ln(f(x^{(i)}))}_a + \underbrace{(1 - y^{(i)}) \ln(1 - f(x^{(i)}))}_b \right]$$

Bei $y = 1$ wird $b = 0$, bei $y = 0$ wird $a = 0$ und es ergibt sich:

$f(x)$	y	$L^{\text{logit}}(D, f)$
1	1	0
1	0	∞
0	0	0
0	1	∞

Ziel: Minimierung der Kostenfunktion, wobei es sich um ein konvexes Optimierungsproblem handelt (d.h. es existiert nur ein globales Minimum). Effiziente Lösung mithilfe numerischer Methoden wie *Gradient Descent* möglich.

Polynomiale Erweiterung sowie **Regularisierung** analog zur *linearen Regression*.

Evaluation: Berechnung der *Konfusionsmatrix* und Einsetzen in die *Klassifikationsmetriken*:

	$y = 1$	$y = 0$
$\text{clf} = 1$	TP	FP
$\text{clf} = 0$	FN	TN

- *Accuracy/Genauigkeit*: $\text{acc}(D, \text{clf}) = \frac{TP+TN}{TP+TN+FP+FN} \rightarrow$ Verhältnis der korrekt klassifizierten Instanzen zu allen Instanzen; bei ungleicher Klassenrepräsentation nicht geeignet
- *Precision*: $\text{prec}(D, \text{clf}) = \frac{TP}{TP+FP} \rightarrow$ wie viele der positiv klassifizierten Instanzen sind tatsächlich positiv?
- *Recall/Sensitivität*: $\text{rec}(D, \text{clf}) = \frac{TP}{TP+FN} \rightarrow$ wie viele Ist-positive Instanzen wurden korrekt klassifiziert?
- *F1-Score*: $F1(D, \text{clf}) = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \rightarrow$ harmonisches Mittel von Precision und Recall (Gesamtqualität)

2.3 Support Vector Machines

Grundidee: Finde die Hyperebene h_0 , die die Klassen trennt und den größtmöglichen Abstand zu den nächsten Trainingsdatenpunkten hat. Die jeweils nächsten Datenpunkte werden als *Support Vectors* bezeichnet.

Anders als bei der logistischen Regression wird bei SVMs die Klassifikationsgrenze *explizit* gelernt und nicht *implizit* berechnet.

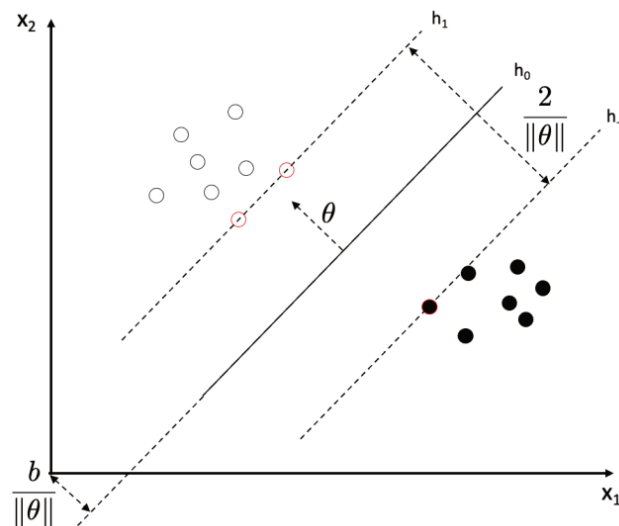


Abbildung 1: Support Vector Machines

- Θ ist der Normalenvektor der Hyperebene
- b ist der Abstand der Hyperebene zum Ursprung

Optimierungsproblem der **Hard-Margin SVM** (Klassen sind linear eindeutig linear trennbar):

$$\begin{aligned} \min \quad & \|\Theta\| \\ \text{s.t.} \quad & y(\Theta^T x - bs) \geq 1 \text{ für } (x, y) \in D \end{aligned}$$

Alternativ: Maximiere den Abstand zwischen den Hyperebenen h_1 und $h_{-1} \rightarrow \max \frac{2}{\|\Theta\|}$

Für die Hyperebene h_0 sowie die parallel verlaufenden Ebenen h_1 und h_{-1} gelten:

$$\begin{aligned} h_0 : \Theta^T x - b &= 0 \\ h_1 : \Theta^T x - b &= 1 \\ h_{-1} : \Theta^T x - b &= -1 \end{aligned}$$

Lineare Separierbarkeit ist in der Praxis selten gegeben. *Hard-Margin SVM* besitzt dann keine zulässige Lösung (undefiniert). Daher wird die **Soft-Margin SVM** verwendet, die Ausreißer zulässt:

$$\min C\|\Theta\|^2 + \underbrace{\frac{1}{m} \sum_{i=1}^m \max(0, 1 - y^{(i)}(\Theta^T x^{(i)} - b))}_{\text{Hinge-Kostenfunktion}(L^{\text{Hinge}})}$$

- $C \in \mathbb{R}^{>0}$ ist ein Regularisierungsparameter ($C\|\Theta\|^2$ ist identisch zum *Tikhonov-Regularisierer*):
 - kleines $C \rightarrow$ ähnliche (gleiche?) Ergebnisse wie *Hard-Margin SVM*
 - großes $C \rightarrow$ Klassifikator wird toleranter ggü. Abweichungen von linearer Separierbarkeit von D
- L^{Hinge} ist die Kostenfunktion, die die Verletzung der Margin-Konstraints bestraft: Bei korrekter Klassifikation wird $L^{\text{Hinge}} = 0$
- $y^{(i)}(\Theta^T x^{(i)} - b)$ ist der Abstand des i -ten Datenpunkts zur Hyperebene

Anwendung auf nicht-lineare Daten durch **Kernel-Trick**: Anstatt (rechenaufwändig) Merkmale zu erweitern (s. lineare/logistische Regression) kann die Transformation implizit durch einen Kernel $k(x^{(i)}, x^{(j)})$ erfolgen. Gebräuchliche Kernel sind:

- $k_{\text{poly-h}}^d(x, x') = (x^T x')^d \rightarrow$ homogener polynomieller Kernel zu Grad $d > 0$
- $k_{\text{poly-i}}^{d,r}(x, x') = (x^T x' + r)^d \rightarrow$ inhomogener polynomieller Kernel zu Grad $d > 0$ mit $r \in \mathbb{R}$
- $k_{\text{rbf}}^\gamma(x, x') = e^{-\gamma\|x-x'\|^2} \rightarrow$ *Gaußscher Kernel*, Radiale Basisfunktion mit $\gamma > 0$

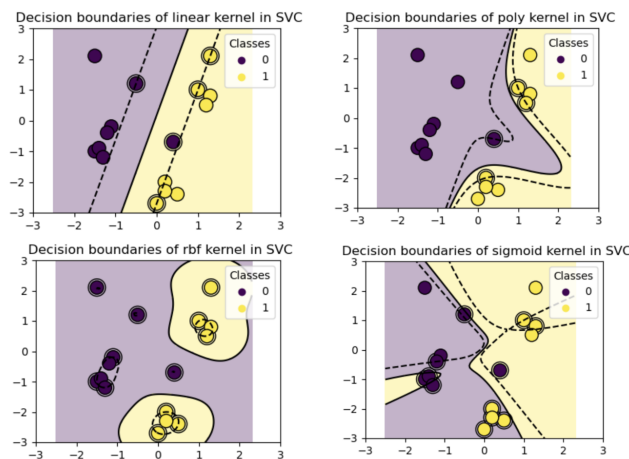


Abbildung 2: Kernel-Unterschiede

2.4 K-Nearest Neighbours

Grundidee: Klassifizierung eines Objekts anhand der Klassenzugehörigkeit seiner k nächsten Nachbarn (aus X_{Train}). Bei mehr als k nächsten Nachbarn wird eine zufällige Auswahl der möglichen Kandidaten gewählt. Wenn Klassenzugehörigkeit nicht eindeutig ist (z.B. $k = 3$ und alle Elemente haben unterschiedliche Klassen) wird eine zufällige gewählt. Parametrisierung:

- Abstandsmessung z.B. per *euklidischer Norm* oder *Manhattan-Norm* $= \sum_{l=1}^n |x_l^i - x_l^j|$ (o.a.).
- Selektion der Klasse anhand der Mehrheitsentscheidung der k nächsten Nachbarn (*maj*) oder anderen (z.B. mit Gewichtung).
- Typische Werte für k liegen im Bereich 1 bis 10, wobei kleinere k zu *Überanpassung* und größere k zu *Unteranpassung* neigen.

Regression: Statt Klassenzugehörigkeit wird der Mittelwert der k nächsten Nachbarn als Schätzung für den Wert des Objekts verwendet.

Vor- und Nachteile:

- + Einfach und intuitiv
- + Keine Annahmen über die Verteilung der Daten
- Langsam bei großen Trainingsdatensätzen
- Sensibel gegenüber Ausreißern
- Wahl von k und Abstandsmessung nicht trivial

Merkmalskalierung: Wichtig bei den meisten ML-Verfahren, da sonst Merkmale mit größeren Werten (Skalen) stärker gewichtet werden (insb. bei Verwendung der Euklidischen Norm). Wichtiger Schritt in der *Datenverarbeitung*: Es geht darum Merkmalsausprägungen zu *normieren*. Gebräuchlichste Variante: *z-Transformation* (bzgl. *Standardisierung*):

$$\text{normiertes Merkmal} \rightarrow \hat{x}_j^{(i)} = \frac{x_j^{(i)} - \bar{x}_j}{\sigma_j}$$

$$\text{Mittelwert} \rightarrow \bar{x}_j = \frac{1}{m} \sum_{i=1}^m x_j^{(i)}$$

$$\text{Standardabweichung} \rightarrow \sigma_j = \sqrt{\frac{1}{m} \sum_{i=1}^m (x_j^{(i)} - \bar{x}_j)^2}$$

→ Merkmale erhalten eine mittlere Ausprägung von 0 und eine Standardabweichung von 1.

2.5 Bayes-Klassifikator

Optimalität: Gegeben einem beobachteten Datensatz D , suche die wahrscheinlichste Hypothese h , die den Wert $P(h|D)$ maximiert.

- **Bayes Theorem:** $P(h|D) = \frac{P(D|h)P(h)}{P(D)}$
- **a posteriori** Wkt. von h : $P(h|D) \rightarrow$ kann im allg. nicht direkt bestimmt werden
- **a priori** Wkt von h : $P(h) \rightarrow$ wie wahrscheinlich ist eine Hypothese \rightarrow Hintergrundwissen oder Annahme von Gleichverteilung
- $P(D) \rightarrow$ Wkt, dass D beobachtet wurde
- $P(D|h) \rightarrow$ Wkt, D zu beobachten, gegeben dass D von h *generiert* wurde \rightarrow wie gut erklärt h den Datensatz D ? Ist einfacher abzuschätzen als $P(h|D)$
- **Maximum-A-Posteriori (MAP)**-Hypothese $\rightarrow h^* = \operatorname{argmax}_{h \in H} P(h|D)$
- **Maximum-Likelihood (ML)**-Hypothese $\rightarrow h^* = \operatorname{argmax}_{h \in H} P(D|h)$

Es lässt sich zeigen, dass das Problem der **(linearen) Regression** mit $\Theta^* = \arg \min \Theta \sum_{i=1}^m (\Theta^T x^{(i)} - y^{(i)})^2$ (d.h. Nutzung der quadratischen Kostenfunktion) eine ML-Hypothese bestimmt, die lineare Regression also nach Bayes'schen Grundsätzen plausibel ist. Dabei wird angenommen, dass die Werte $y^{(i)}$ eines Datensatzes D durch den wahren Wert der Merkmalsausprägung $\hat{y}^{(i)}$ und einen Fehler $\epsilon^{(i)}$ bestimmt sind. Bzgl. $\epsilon^{(i)}$ wird üblicherweise Normalverteilung angenommen (Herleitung: Skript S. 59).

Nachteile des (normalen) Bayes Klassifikators:

- jede mögliche Merkmalsausprägung (Kombination aus mehreren Merkmalen) muss im Trainingsdatensatz vorhanden sein, da Wahrscheinlichkeit sonst undefiniert ($\frac{0}{0}$) ist
- der gesamte Datensatz D muss vorgehalten werden \rightarrow Höherer Speicherbedarf und längere Berechnungszeiten

Diese Nachteile lassen sich mittels **Naivem Bayes Klassifikators** lösen. Dieser geht davon aus, dass die einzelnen Merkmalsausprägungen bedingt unabhängig voneinander sind:

$$\text{clf}_D^{\text{NaiveBayes}}(x) = \arg \max_{c \in Z} P(c|D)P(x_1|c, D)P(x_n|c, D) \dots P(x_n|c, D)$$

- $P(c|D) \rightarrow$ Für jede Klasse c : Wie oft kommt diese in D vor (Wkt)?
- $P(x_i|c, D) \rightarrow$ Für eine gegebene Klasse c : Einzelwkt. für $x_i = \dots$

Verwendung *kontinuierlicher Merkmale* möglich, indem die direkt aus den Daten berechnete Verteilung $P(x_i|c, D)$ durch eine abgeschätzte Dichte $p(x_i|c, D) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x_i-\mu)^2}{2\sigma^2}}$ ersetzt wird.

2.6 Entscheidungsbäume

Eigenschaften:

- Intuitiv und regelbasiert nachvollziehbar
- *Entscheidungsbaum* T mit Blattknoten (auch: *Klassifikationsknoten*), inneren Knoten (auch: *Entscheidungsknoten*) und Kanten (E) stellt einen (abwärts) gerichteten Baum mit Wurzel r dar.
- Jeder Knoten stellt eine Fallunterscheidung für den Wert eines Merkmals eines gegebenen Datenpunktes dar.
- Für jeden Fall gibt es einen eindeutigen Nachfolgeknoten und die Blätter des Baumes enthalten die möglichen Klassifikationen des betrachteten Datenpunktes.

Lösung üblicherweise nicht durch Lösen eines Optimierungsproblems sondern durch dedizierte prozedurale Algorithmen wie **ID3** (Iterative Dichotomiser 3) oder **C4.5**:

- Basieren auf dem Prinzip der *top-down induction of decision trees* (TDIDT) \rightarrow baut Entscheidungsbaum rekursiv von der Wurzel beginnend auf
- Abbruchbedingungen \rightarrow alle Beispiele in der betrachteten (Teil-)Menge
 - sind von derselben Klasse
 - haben identische Merkmalsausprägungen \rightarrow wähle am häufigsten vorkommende Klasse
- *C4.5-Algorithmus* ist eine Weiterentwicklung von ID3, enthält jedoch folgende Optimierungen:
 - ID3 tendiert bei leicht verrauschten Daten zur Überanpassung (d.h. sehr lange Pfade); C4.5 enthält einen Nachverarbeitungsschritt, der weniger relevante Entscheidungsregeln erkennt und den Baum kürzt (sog. *pruning*)
 - bessere Behandlung kontinuierlicher Merkmale
 - optimierte (skalierte) Version des *Informationsgewinns* (s.u.) \rightarrow unten dargestellte Version bevorzugt Merkmale mit einer hohen Anzahl an Merkmalsausprägungen

Merkmal, nach dem die jeweils nächste Aufteilung erfolgt soll so gewählt werden, dass der Baum möglichst klein wird \rightarrow wähle Merkmal mit höchstem Informationsgewinn (IG) unter Verwendung der *Entropie*.

- **Entropie:** $H(D) = - \sum_{i=1}^k \text{relative Häufigkeit}_i \times \log_{10}(\text{relative Häufigkeit}_i) \rightarrow$ je höher $H(D)$, desto gleichverteilter treten die Klassen auf (z.B. = 0, wenn alle Element einer einzelnen Klasse angehören; dann steht Klassifikation fest).
- **bedingte Entropie:** Was passiert mit der Entropie, wenn ein bestimmtes Merkmal ausgewählt wird? $H(D, i) = \sum \text{rel. Häufigkeit der Merkmalsausprägung}_i \times \text{Entropie der Teilmenge}_i$
- **Informationsgewinn:** $IG(D, i) = H(D) - H(D|i) \rightarrow$ gibt an, wie sehr die Merkmalsauswahl i die Daten in D nach den Klassen (vor-)sortiert. Je höher der Informationsgewinn, desto besser klassifiziert das Merkmal i den Datensatz D

3 Unsupervised Learning

3.1 K-Means Clustering

Idee: Eine Datenmenge E so in Teilmengen zu partitionieren, dass Datenpunkte in jedem E_i *ähnlich* zueinander sind, wohingegen Datenpunkte in unterschiedlichen E_i *unterschiedlich* sind.

Klassischer Algorithmus ist der **K-Means-Algorithmus**, der große Ähnlichkeit zum *KNN-Algorithmus* hat. Naive Implementierung mittels *Lloyds-Algorithmus*:

- Gegeben eine Datenmenge E und eine Anzahl k an Clustern $\rightarrow k$ muss zwingend vorgegeben werden
- Iterative Berechnung, beginnend bei zufälligen k Zentren (*Zentroiden*):
 - Berechnet die euklidische Distanz zwischen jedem Datenpunkt und jedem Zentroiden
 - Weist jeden Datenpunkt dem nächstgelegenen Zentroiden zu
 - Berechnet die neuen Zentroiden als Mittelwert der Datenpunkte in jedem Cluster
 - Wiederholt die Schritte 2 und 3, bis sich die Zentroiden nicht mehr verändern
- Wahl von k und der Initialisierung der Zentroiden sind kritisch und beeinflussen das Ergebnis \rightarrow Wiederholte Ausführung mit unterschiedlichen Parametern
- *Euklidische Distanz* \rightarrow Hohe Anfälligkeit für unterschiedliche Skalen und Ausreißer; Normalisierung notwendig (z.B. durch *z-Transformation*)

externe Evaluation: Clustering wird auf zusätzlichem Testdatensatz evaluiert, der die tatsächlichen Clusterzugehörigkeiten enthält. Aufwendig, da Annotation meistens manuell erfolgen muss. \rightarrow Verwendung der gleichen Metriken wie bei überwachten Lernen (Genauigkeit, F1,...)

interne Evaluation: Orientierung am eigentlichen Optimierungsziel (finde Zentroiden mit minimalen quadrierten Distanzen aller Datenpunkte zu den ihnen zugewiesenen Zentroiden). Formal definiert durch *Trägheitsmaß* (engl. *inertia*):

$$\text{inertia}(\text{cluster}, E, m_1 \dots m_k) = \sum_{x \in E} \|x - m_{\text{cluster}(x)}\|^2$$

- finden von cluster und $m_1 \dots m_k$, so dass *inertia* minimal ist, ist *NP-schwer*, weshalb K-Means auch nur versucht eine Annäherung an das Optimum zu erreichen.
- tatsächliche Wert von *inertia* nicht leicht zu interpretieren, da hohe Abhängigkeit von der Skalierung \rightarrow vom Trägheitswert lässt sich keine Aussage zur Güte des Clusterings ableiten
- Vergleich von Trägheitswerten unterschiedlicher Clusterings ist möglich \rightarrow Clusterings müssen über die gleiche Clusteranzahl verfügen
- Je mehr Cluster, desto geringer ist üblicherweise die Trägheit
- Wenn $k = m$ (ein Cluster pro Datenpunkt), dann ist die Trägheit 0
- Verwendung des Trägheitsmaßes, um die optimale Clusteranzahl zu bestimmen \rightarrow *Elbow-Methode*: Berechnung der Trägheit für unterschiedliche Clusteranzahlen und Bestimmung des Knickpunktes

K-Means++: Verbesserte Initialisierung der Zentroiden, um Konvergenz zu beschleunigen: Wähle Zentroiden im 1. Durchlauf wie bisher aus vorhandenen Datenpunkten. Alle folgenden Zentroiden werden mit der gewichteten Wahrscheinlichkeit ihrer Distanz zu bereits gewählten Zentroiden ausgewählt. Ergebnis: Der Trägheitswert des errechneten Clusterings ist maximal um einen Faktor $O(\log k)$ größer als der Trägheitswert des optimalen Clusterings.

3.2 Hierarchisches Clustering

Idee: Es gibt keine allgemeingültige Definition der *korrekten* Clusteranzahl; je nach Anwendung können für denselben Datensatz verschiedene Clusterzahlen Sinn ergeben. Durch hierarchische Aufteilung der Datenmenge in einem **Dendrogramm**

- besserer Einblick in Zusammenhänge und Unterstützung bei der Findung der optimalen Clusterzahl.
- jeder innere Knoten v mit Kindern $v_1 \dots v_k$ repräsentiert ein Cluster, das durch die Vereinigung der Cluster $v_1 \dots v_k$ entsteht.
- Blätter repräsentieren die einzelnen Datenpunkte (als eigener Cluster).
- Wurzel enthält das gröbste Clustering (alle Datenpunkte in einem Cluster).
- *Quantitatives Dendrogramm* erlaubt die quantitative Abschätzung zur Plausibilität verschiedener Hierarchiestufen.

Agglomeratives Clustering (bottom-up): Man beginnt mit jedem Datenpunkt als eigenem Cluster und fügt iterativ die zwei ähnlichsten Cluster zusammen, bis nur noch ein Cluster übrig ist. Z.B. Verwendung des **Single-Link-Clustering**:

- Berechnung der Distanz zwischen allen Clustern \rightarrow Je nach Distanzfunktion andere Ergebnisse; hier: $D_{\text{single}}(E_1, E_2) = \min_{x_1 \in E_1, x_2 \in E_2} \|x_1 - x_2\|$ (=minimaler euklidischer Abstand), Merkmale müssen entsprechend skaliert sein
- Zusammenfassung der beiden ähnlichsten Cluster \rightarrow bei Uneindeutigkeit zufällige Wahl
- Wiederholung der Schritte 1 und 2, bis nur noch ein Cluster übrig ist
- liefert relativ *langgezogene* Cluster und eignet sich für entsprechende *kettenförmige* Clusterstrukturen

Divisives Clustering (top-down): Man beginnt mit allen Datenpunkten in einem Cluster und teilt iterativ das Cluster in zwei Teilmengen, bis jeder Datenpunkt in einem eigenen Cluster ist. Z.B. Verwendung des **Divisive Analysis Clustering (DIANA)**: deutlich aufwendiger als agglomeratives Clustering, da alle möglichen Clusterkombinationen betrachtet werden müssen. Deshalb wenig Praxisrelevanz.

3.3 Assoziationsregeln

Idee: Entdeckung häufig zusammen auftretender Mengen von Elementen, um Zusammenhänge zwischen Attributen zu identifizieren.

Metriken (liegen stets zwischen 0 und 1, sollten beide möglichst hoch sein):

- **Support**: Anteil der Transaktionen, in denen die Regel gilt \rightarrow wie *oft* kann eine Regel angewendet werden?
- **Konfidenz**: Anteil der Transaktionen, in denen A und B gemeinsam vorkommen, an den Transaktionen, in denen A vorkommt $\text{conf}_F(A \Rightarrow B) = \frac{\text{support}(A \cup B)}{\text{support}(A)}$ \rightarrow wie *gut* bildet die Regel den Zusammenhang zwischen Prämisse und Konklusion ab?

A-Priori-Algorithmus: Idee: Ober-Mengen nicht häufiger Teilmengen müssen ebenfalls nicht häufig sein und können ignoriert werden. *Bottom-Up*-Ansatz, um Kombinationen von Attributen zu finden, die die Mindestunterstützung (*minsupp* und *minconf*) erfüllen.

FP-Growth-Algorithmus: Weiterentwickelte Implementierung, die auf *Frequent Pattern Trees* basiert und im Durchschnitt signifikant weniger Kandidaten berücksichtigt \rightarrow damit signifikant schneller als oben

3.4 Anomalieerkennung

Idee: Identifikation von Datenpunkten, die sich signifikant von der Mehrheit der Daten unterscheiden.

Wenn abnormale Datenpunkte bekannt sind, kann man das Problem als überwachtes Lernproblem betrachten. Ansonsten als unüberwachtes Lernproblem.

Dichteabschätzung einer Normalverteilung: (Maximum-Likelihood-Methode)

- Ein Datenpunkt ist abnormal, wenn die Wahrscheinlichkeit $p^E(x) < \epsilon$, mit dem Schwellwert $\epsilon \in [0, 1]$
- Annahme 1: Merkmale sind *unabhängig* voneinander, d.h. $p^E(x) = p_1^E(x_1) \cdot \dots \cdot p_n^E(x_n)$
- Annahme 2: Merkmalsausprägungen eines jeden Merkmals x_i sind *normalverteilt*, d.h. es gilt

$$p_i^E(x_i) = \frac{1}{\sqrt{2\pi}\sigma_i} e^{-\frac{(x_i - \mu_i)^2}{2\sigma_i^2}} \text{ mit Mittelwert } \mu_i = \frac{1}{m} \sum_{j=1}^m x_i^{(j)} \text{ und Varianz } \sigma_i^2 = \frac{1}{m} \sum_{j=1}^m (x_i^{(j)} - \mu_i)^2$$

3.5 Hauptkomponentenanalyse/Principal Component Analysis (PCA)

Methode zur *Dimensionsreduktion* und *Datenkompression* von hochdimensionalen Daten, so dass möglichst wenig Informationen verloren gehen. Insb. sinnvoll, da in der Praxis Merkmalsausprägungen oft korrelieren.

Zwingende Voraussetzung: Standardisierung der Datenpunkte (beim supervised Learning sind Klassen nicht zu standardisieren), d.h. Mittelwert der Daten auf Null und Varianz auf Eins (z.B. mittels *z-Transformation*).

Funktionsweise: Suche nach einer neuen Basis (einem Unterraum), die die Varianz der Datenpunkte maximiert. Die Basisvektoren sind die *Hauptkomponenten* (PCs), die orthogonal zueinander sind. Am Beispiel $\mathbb{R}^2 \mapsto \mathbb{R}^1$: finde eine Gerade, so dass die quadrierte Distanz der Datenpunkte zur Geraden minimal ist. Diese Gerade geht durch den Nullpunkt (\rightarrow deshalb ist Standardisierung wichtig!)

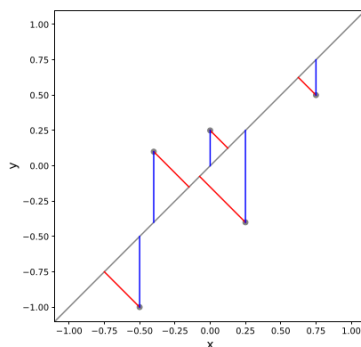


Abbildung 3: Zielfunktion von PCA minimiert die Summe der quadrierten Distanzen, orthogonal zur Geraden vs. Zielfunktion der linearen Regression, die Summe der vertikalen Abstände minimiert. $\text{PCA} \neq \text{linReg}$!

Bei der *linearen Regression* geht es um Funktionsapproximation, bei PCA um Dimensionsreduktion.

Berechnung: Hauptkomponenten entsprechen den Eigenvektoren der Kovarianzmatrix der Datenpunkte. Die Eigenwerte geben die Varianz entlang der Hauptkomponenten an.

- Kovarianzmatrix aufstellen: $\text{Cov}(E) = \frac{1}{m} \sum_{i=1}^m x^{(i)}(x^{(i)})^T$
- Kovarianzmatrix ist symmetrisch und positiv definit \rightarrow Diagonalisierung möglich; es existieren genau n reale Eigenwerte $\lambda_1 \dots \lambda_n$ und Eigenvektoren $v_1 \dots v_n$
- Eigenvektoren sind orthogonal zueinander und normiert \rightarrow bilden die Hauptkomponenten

Evaluation:

- Verwendung der Evaluationsmaße der eigentlichen Lernaufgabe; z.B. F1-Maß für Klassifikation \rightarrow aufwendig, da Datensätze verschiedener Dimensionalität gebildet und verglichen werden müssen
- besser: Messung des Verlustes an Varianz (wie verstreut sind die Datenpunkte E im Raum?) durch Dimensionsreduktion \rightarrow Wähle die Dimensionalität, bei der der Verlust an Varianz *nicht zu hoch* ist *rightarrow* hier muss Model nicht neu gelernt werden.
- Varianz der Datenpunkte: $\text{var}(E) = \frac{1}{m} \sum_{i=1}^m \|x^{(i)}\|^2$
- Retained Variance: $\text{retVar}(E^{\text{compressed}}, E) = \frac{\text{var}(E^{\text{compressed}})}{\text{var}(E)} \rightarrow$ Anteil der Varianz von E , der in $E^{\text{compressed}}$ erhalten bleibt; es gilt stets $\text{var}(E^{\text{compressed}}) \leq \text{var}(E)$.
- Wähle die Dimensionalität so klein wie möglich, so dass $\text{retVar}(E^{\text{compressed}}, E) \geq \alpha$ für $\alpha \in [0, 1]$. Typischerweise wird $\alpha \in [0.9, 0.99]$ gewählt.

4 Reinforcement Learning

→ Modellierung der Dynamik der Umgebung, in der ein Agent eingebettet ist.

Offline-Lernen: Lernen aus einer Menge von Beispielen, die bereits gelabelt sind; Der Algorithmus hat ein korrektes Modell der Umgebung, der Auswirkungen von Aktionen und den Belohnungen.

Online-Lernen: Lernen während der Interaktion mit der Umgebung; Der Algorithmus hat keinerlei Vorwissen.

4.1 Markov-Entscheidungsprozesse

Für alle Varianten des *Reinforcement Learning* werden *Markov-Entscheidungsprozesse* (MEP) als Mittel zur Formalisierung dynamischer Prozesse (Modell der Umgebung) verwendet.

Der *Discountfaktor* $\gamma \in [0, 1]$ bestimmt beim *diskontierten Nutzen* $U_D^\gamma(e) = \sum_{i \geq 0} \gamma^i R(s_i, a_i, s_{i+1})$, wie stark zukünftige Belohnungen gewichtet werden. Bei kleinerem γ wird die Zukunft weniger stark gewichtet, d.h. es werden Strategien bevorzugt, die schnell hohe Belohnungen erhalten. Üblich: γ wird echt kleiner, aber nahe 1 gewählt (z.B. 0.9 oder 0.99) → gewährleistet, dass der Nutzen $U_D^\gamma(e)$ stets endlich ist, selbst bei unendlich langen Episoden mit positiver Wahrscheinlichkeit.

$U_D^\gamma(\pi)$ ist der erwartete durchschnittliche Nutzen aller aus π generierten initialen Episoden, gewichtet nach deren Wahrscheinlichkeit.

Ein Markov-Entscheidungsprozess kann durch ein Zustandsübergangsdiagramm formalisiert werden.

Unterscheidung von *konstanten Strategien* (sehen für jeden Zustand s eine feste Aktion a vor) und *probabilistischen Strategien* (halten für jeden Zustand s eine Wahrscheinlichkeitsverteilung über die auszuwählende Aktionen vor).

4.2 Passives Reinforcement-Learning

Test

4.3 Aktives Reinforcement-Learning

Test

5 Deep Learning

5.1 Künstliche Neuronale Netze (ANN)

Können eingesetzt werden für

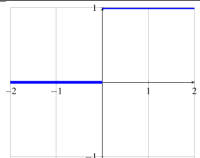
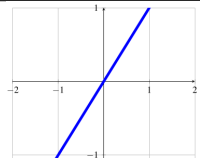
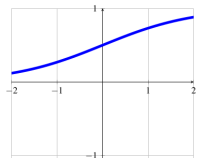
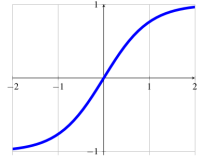
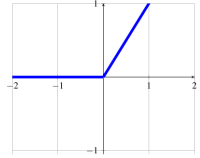
- überwachtes Lernen
- unüberwachtes Lernen
- Reinforcement Learning

Idee: Komplexe Lernaufgabe wird durch viele versteckte Schichten in Teilaufgaben zerlegt. Vorteil dabei ist, dass die beim Lernalgorithmus beim Lernen der Gewichte auch selbst entscheidet, welche Zwischenkonzepte die Schichten repräsentieren → *feature-engineering* (manuelle Definition der Merkmale) entfällt.

Feedforward-Netzwerk: *Input Layer* nimmt Eingabewerte entgegen, *Output Layer* gibt Ausgabewerte aus. Dazwischen liegen *Hidden Layer*, die die Eingabewerte transformieren. Ab 2 Hidden-Layern spricht man von einem *Deep Neural Network*. Die einzelnen Neuronen sind in Schichten angeordnet, wobei jedes Neuron mit jedem Neuron der vorherigen Schicht *gewichtet* verbunden ist. Über die *Aktivierungsfunktion* (nimmt die gewichteten Werte vorheriger Neuronen sowie einen *Bias-Wert* entgegen) wird die Ausgabe eines Neurons bestimmt. Evaluert wird das Netzwerk über eine *Verlustfunktion*, die den Fehler zwischen den Ausgaben des Netzwerks und den tatsächlichen Ausgaben berechnet.

Aktivierungsfunktionen:

Die *Aktivierungsfunktion* ist typischerweise in allen *Hidden-Layern* gleich; im Ausgabebereich aber anwendungsabhängig und üblicherweise nicht identisch mit denen der Hidden-Layer (Regression: Ausgabe muss gesamte reellen Zahlen als Wertebereich haben; Klassifikation: Bool'sche Ausgabe (wenn binär) oder eine Wahrscheinlichkeit, ggf. auch mehrere Ausgabeneuronen für einzelne Klassen).

Funktion	Formel	Ableitung	Eigenschaften	Verlauf
Threshold	$h^{\text{thresh}}(x) = \begin{cases} 1 & \text{für } x \geq 0 \\ 0 & \text{für } x < 0 \end{cases}$	nicht differenzierbar	Praxis (-)	
Identität	$h^{\text{id}}(x) = x$	$h^{\text{id}}(x)' = 1$	Praxis (-); nur im Output bei Regression	
Sigmoid	$h^{\text{logit}}(x) = \frac{1}{1+e^{-x}}$	$h^{\text{logit}}(x)' = h^{\text{logit}}(x)(1 - h^{\text{logit}}(x))$	$h^{\text{logit}}(x) \in [0, 1]$ → bin. Klassifikat.	
Hyperbol. Tangent	$h^{\text{tanh}}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$h^{\text{tanh}}(x)' = 1 - (h^{\text{tanh}}(x))^2$	$h^{\text{tanh}}(x) \in [-1, 1]$	
Rectified Linear Unit	$h^{\text{ReLU}}(x) = \max(0, x)$	$h^{\text{ReLU}}(x)' = \begin{cases} 0 & \text{für } x < 0 \\ 1 & \text{für } x \geq 0 \end{cases}$	$h^{\text{ReLU}}(x) \in [0, \infty)$	
Softmax			$h^{\text{softmax}}(x_i) \in [0, 1]$ $\sum_{i=1}^n h^{\text{softmax}}(x_i) = 1$ → Mehrklassenklassifikation	

Kosten-/Verlustfunktionen:

Funktion	Formel	Ableitung	Eigenschaften
Quadratischer Fehler	$L^{\text{qF}}(D, f) = \sum_{i=1}^m (f(x^{(i)}) - y^{(i)})^2$	$L'(D, f) = 2(f(x) - y)$	für h^{id}
Logistische Kosten	$L^{\text{logit}}(D, f) = -\sum_{i=1}^m [y^{(i)} \ln(f(x^{(i)})) + (1 - y^{(i)}) \ln(1 - f(x^{(i)}))]$	$L'(D, f) = f(x) - y$	für h^{logit} oder h^{ReLU}

Bei Verwendung der Identitätsfunktion $h^{\text{id}} = x$ als Aktivierungsfunktion und dem quadratischen Fehler L^{qF} als Verlustfunktion ist die Ausgabe äquivalent zu der optimal angepassten linearen Funktion für die lineare Regression.

Bei Verwendung der Sigmoid-Funktion h^{logit} als Aktivierungsfunktion und der logistischen Kostenfunktion L^{logit} ist die Ausgabe äquivalent zu dem optimalen Modell der logistischen Regression.

Backpropagation:

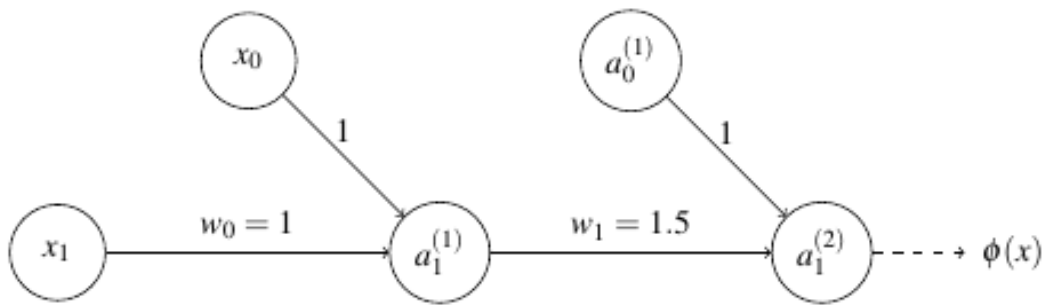


Abbildung 4: Ausgangsbeispiel mit $x_1 = 0.1$, $x_2 = 0.2$, $a_0^{(1)} = 0.05$, $y = 0.3$, $\text{act} = \text{ReLU}$, L^{logit} und $\gamma = 0.1$

1. Forward-Pass: Berechne die Ausgabe des Netzes für einen gegebenen Eingabevektor $x \rightarrow$ Netzwerk von Input bis Output unter Berücksichtigung von Gewichten, Bias und Aktivierungsfunktionen durchrechnen.

$$\begin{aligned} z_1^{(1)} &= w_0 x_1 + 1 x_0 = 1 \cdot 0.2 + 0.1 = 0.3 \rightarrow a_1^{(1)} = h^{\text{relu}}(z_1^{(1)}) = \max[0, 0.3] = 0.3 \\ z_1^{(2)} &= w_1 a_1^{(1)} + 1 a_0^{(1)} = 1.5 \cdot 0.3 + 0.05 = 0.5 \rightarrow a_1^{(2)} = h^{\text{relu}}(z_1^{(2)}) = \max[0, 0.5] = 0.5 = \Phi(x) \end{aligned}$$

2. Kosten- und Fehlerberechnung: Berechne den Fehler des Netzes durch die Verlustfunktion L ; beginne beim Output-Layer und rechne nach vorne.

$$\text{cost}(y, a_1^{(2)}) = -(y \ln(a_1^{(2)}) + (1 - y) \ln(1 - a_1^{(2)})) = -(0.3 \cdot -0.693 + 0.7 \cdot -0.693) = 0.693$$

Berechnung des Fehlers im linearen Anteil des Output-Layers durch:

$$\delta(z_j^{(k)}) = \frac{\partial \text{cost}(y, a_1^{(2)})}{\partial z_1^{(2)}}$$

\rightarrow Aktivierungsfunktion in Kostenfunktion einsetzen und nach $z_1^{(2)}$ ableiten.

Wenn Kostenfunktion = L^{logit} (Logistische KF) und Aktivierungsfunktion = h^{logit} (Sigmoid), gilt $\delta(z_i^{(l)}) = a_i^{(l)} - y_i$.

Der Fehler im linearen Anteil $\delta(z_1^{(2)})$ der Hidden Layer ist gegeben durch:

$$\delta(z_j^{(k)}) = \text{act}'(z_j^{(k)}) \cdot \sum_{i=1}^{n_{k+1}} \delta(z_i^{(k+1)}) \cdot w_{ij}^{(k+1)}$$

Fehler eines Neurons berechnet sich aus dem Produkt von

- linearer Wert des aktuellen Neurons ($z_j^{(k)}$) eingesetzt in die Ableitung der Aktivierungsfunktion

- Summe der Fehler der Neuronen der nächsten Schicht multipliziert mit den Gewichten der Verbindung zu diesen Neuronen

3. Neuberechnung der Gewichte: Berechne die neuen Gewichte durch Anpassung der alten Gewichte um den Fehler zu minimieren.

$$w_{ij}^{(k)\text{neu}} = w_{ij}^{(k)\text{alt}} - \gamma \cdot \delta(z_j^{(k)}) \cdot a_i^{(k-1)}$$

Neues Gewicht ergibt sich aus dem alten Gewicht abzüglich des Produkts aus

- Lernrate γ
- Fehler des Neurons
- Ausgabe des Neurons der vorherigen Schicht

4. Wiederhole 1.-3. Schritt: Forward-Pass, Neuberechnung des Fehlers und ggf. weitere Backpropagation zur Modellverbesserung

Auswahl des Trainingsdatensatzes: Gradient Descent mit gesamtem Datensatz kann u.U. sehr langsam sein, wenn D sehr groß ist. Alternativen:

- *Stochastic Gradient Descent (SGD)*: Berechne den Fehler und die Gewichtsanzpassung anhand eines zufällig gewählten Datenpunktes
 - Vorteil: schnelleres Lernen
 - Nachteil: ungenaue Schätzung des Gradienten \rightarrow nächstes lokales Minimum wird nicht zielstrebig angesteuert, Gesamtkosten können während des Lernens stark schwanken, keine Parallelisierbarkeit
- *Mini-Batch Gradient Descent (MBGD)*: Berechne den Fehler und die Gewichtsanzpassung anhand einer zufällig gewählten Teilmenge des Datensatzes. Kompromiss zwischen SGD und GD mit geringerem Fluktuationsgrad. Parallelisierbarkeit indem die Batchgröße entsprechend der Anzahl zu Verfügung stehender Rechenkerne gewählt wird.

Auswahl der Aktivierungsfunktion: Ableitung der Aktivierungsfunktion ist Bestandteil des Algorithmus. Bestimmte Funktionen können zu Problemen führen, wenn bei bestimmten (x)-Werten die Ableitung gegen 0 geht (z.B. Sigmoid-Funktion bei $x \geq 3$ oder $x \leq -3 \rightarrow < 0.1$; sog. *vanishing gradient problem*). Ähnlich: *Exploding Gradient Problem* bei Werten > 1 . Lösung: ReLU-Aktivierung, da für $x > 0$ die Ableitung von $h^{\text{relu}} = 1$ und damit gedeckelt. Nachteil: für $x < 0$ ist die Ableitung stets 0 \rightarrow Lösung durch Verwendung von *plus*.

5.2 Convolutional Neural Networks (CNN)

Spezielle Form des künstlichen neuronalen Netzes, das besonders für die Verarbeitung von Bildern oder Zeitreihen geeignet ist. Ein klassisches *Feedforward*-Netzwerk ist für die Verarbeitung von Bildern ungeeignet, da es die räumliche Struktur der Daten ignoriert (da ein Bild z.B. eine Tür überall haben kann, müsste es entsprechend komplex trainiert werden). CNNs hingegen nutzen die räumliche Struktur von Bildern aus, indem sie spezielle Schichten verwenden, die als Filter fungieren.

Verwendung der Faltungsoperation (*):

$$\begin{aligned} \text{kontinuierlich: } (i * k)(x) &= \int_{-\infty}^{\infty} i(y)k(x-y)dy \\ \text{diskret: } (I * K)(x) &= \sum_{m=-\infty}^{\infty} I(m)K(n-m) \end{aligned}$$

mit i/I als Eingabe und k/K als Filter (auch: Kernelfunktion). Die Ausgabe $I * K$ heißt auch *Feature Map*. Einfaches Anwendungsbeispiel: *Denoising* von verrauschten Daten.

Intuition: Man schiebt den Filter (eine z.B. 3×3 -Matrix) über das Bild (Matrix mit Graustufenwerten oder 3 RGB Matrizen) und berechnet das Skalarprodukt zwischen Filter und Bildausschnitt. Dieses Skalarprodukt wird anschließend in eine Aktivierungsfunktion (z.B. ReLU) gegeben.

Vorteil ggü. ANN:

- relativ wenig Kanten zwischen den einzelnen Schichten (*Sparse Interaction*), da jedes Neuron nur mit einem kleinen Ausschnitt der vorherigen Schicht verbunden ist.
- Es gibt eine ganze Menge an Kanten zwischen den einzelnen Schichten die das gleiche Kantengewicht haben (Stichwort: *Parameter Sharing*).

Padding:

- *Valid*: keine Randbehandlung, Filter wird nur über die Bildmatrix geschoben, wenn er komplett auf der Bildmatrix liegt → Ausgabe (Feature Map) kleiner als Eingabe.
- *Half/Same*: Um die gesamte Matrix wird ein Rahmen (z.B. beim *Zero-Padding* mit Werten 0) gezogen, sodass die Ausgabe die gleiche Größe wie die Eingabe hat.
- *Full*: Padding wird hinzugefügt, sodass die Ausgabe die Größe der Eingabe plus die Größe des Filters minus 1 hat.

Stride/Schrittweite: Bestimmt, um wieviele Pixel/Positionen in der Matrix der Filter verschoben wird (horizontal und vertikal). Je höher der Stride, desto kleiner wird die Feature-Map.

Pooling: Reduziert die Dimensionalität der Feature-Map, indem es den maximalen Wert (Max-Pooling) oder den Durchschnittswert (Average-Pooling) eines Fensters berechnet. Wird i.d.R. abwechselnd zur Faltung angewandt.

Letzter Teil eines CNN ist ein normales ANN (Feedforward-Netzwerk), das die Ausgabe der Faltungsschichten verarbeitet und die eigentliche Klassifikation vornimmt.

5.3 Recurrent Neural Networks

Recurrent Neural Networks (RNN) sind eine spezielle Form des künstlichen neuronalen Netzes, die besonders für die Verarbeitung von Sequenzen (z.B. Zeitreihen oder Wortvorhersagen) geeignet sind.

One-Hot-Kodierung: Vektorisierung von Kategorien, sodass jede Kategorie durch einen Vektor repräsentiert wird, der nur eine 1 an der Stelle der Kategorie hat, sonst 0, z.B.

Kategorien: Apfel, Banane, Birne

$$\text{One-Hot-Kodierung: Apfel} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \text{Banane} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \text{Birne} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

Eigenschaften:

- Kategorien/Vokabular ist vollständig definiert → feste Anzahl an Kategorien
- Anzahl an Eingaben (z.B. Wörter) kann variieren → anders als Feedforward-Netzwerke mit fixer Anzahl
- Wie bei CNN: Lokalität und Parameter Sharing, spätere Vorhersagen werden konzeptionell so behandelt, wie frühere Vorhersagen

Aufbau:

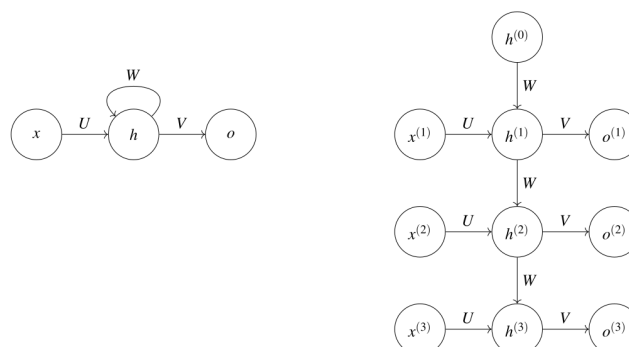


Abbildung 5: Übersicht eines RNN

Beginnend bei einem initialen Zustand $h^{(0)}$ wird für jede Eingabe $x^{(i)}$ mittels Parametermatrix U ein neuer (*hidden*) Zustandsvektor $h^{(i)}$ berechnet. Dieser Vektor wird zum einen zur Bestimmung einer (Zwischen-)Ausgabe $o^{(i)}$ (mittels Gewichtsmatrix V) und zum anderen zur Bestimmung eines neuen (versteckten) Zustands $h^{(i+1)}$ (mittels Gewichtsmatrix W und Aktivierungsfunktion $\text{act}()$) verwendet. h fungiert hierbei als eine Art *Gedächtnis*:

$$h^{(i)} = \text{act}(Ux^{(i)} + Wh^{(i-1)})$$

$$o^{(i)} = \text{act}(Vh^{(i)})$$

Probleme:

- *Vanishing Gradient*: Gradienten werden bei der Rückwärtspropagierung durch Wiederholte Multiplikation mit W so klein, dass sie kaum noch Einfluss auf die Gewichts Anpassung haben.
- *Long-Term Dependencies*: RNNs haben Schwierigkeiten, lange Abhängigkeiten zu lernen, da der Gradient bei der Rückwärtspropagierung durch viele Schichten hindurchgehen muss (s.o. wdh. Multipl. mit W).

Lösung der Probleme zB. über LSTM (Long Short-Term Memory; spezielle RNN-Architektur).

5.4 Repräsentationen

5.4.1 Autoencoder

Feedforward-Netzwerk, das darauf trainiert wird, die Eingabe zur Ausgabe zu kopieren. Kodierer und Dekodierer sind antisymmetrisch aufgebaut (Anzahl der Eingabeneuronen im Kodierer entspricht Anzahl der Ausgabeneuronen im Dekodierer). Die dazwischen liegende Schicht heißt *Flaschenhals*, wenn der Autoencoder *undercomplete*, d.h. $\hat{n} < n$ ist.

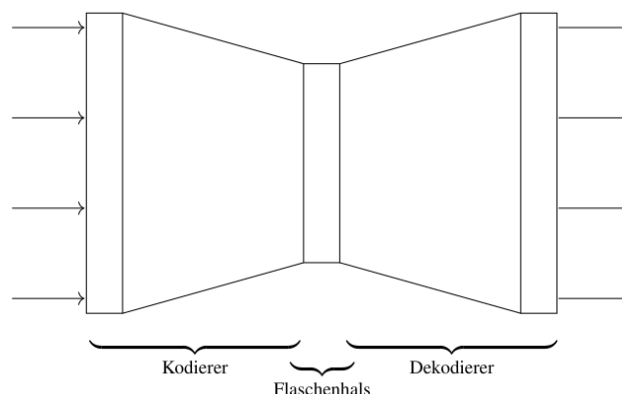


Abbildung 6: Autoencoder

Kompression: Kodierer mit Funktion f und Dekodierer mit Funktion g so trainieren, dass der Flaschenhals (undercomplete Autoencoder mit $\hat{n} < n$) $g \circ f$ einen Wert x_i möglichst Verlustfrei wieder ausgibt. Kodierer wird dabei genutzt Daten zu komprimieren, Dekodierer um Daten zu dekomprimieren. Sind alle Aktivierungsfunktionen linear, hat der Autoencoder die gleiche Kompressionsstruktur wie bei der PCA.

Sparse Autoencoder (SAE): Flaschenhals hat $\hat{n} > n$ und ist damit overcomplete \rightarrow kann z.B. eingesetzt werden, um Ausgabe besser zu generalisieren. Damit potentiell alle Neuronen im Flaschenhals überhaupt genutzt werden, wird ein Regularisierungsparameter λ in der Loss-Funktion eingeführt ($L^{\text{sae}}(D, g \circ f) = L^{\text{auto}}(D, g \circ f) + \lambda \sum_{i=1}^m \|f(x^{(i)})\|$)

Denoising Auto Encoder (DAE): Autoencoder wird mit verrauschten Eingabedaten trainiert (Erzeugt zB. durch Gauss), um die ursprünglichen Daten wiederherzustellen \rightarrow Auch: Datengenerierung als wichtige Ausgabe.

5.4.2 Generative Adversarial Networks (GANs)

GANs bestehen aus zwei separaten *Feedforward*-Netzwerken: Einem *Generator* und einem *Diskriminator*. Der Generator erzeugt Daten (z.B. Bilder), die dem Diskriminator vorgelegt werden. Der Diskriminator versucht zu entscheiden, ob die Daten vom Generator oder aus dem Trainingsdatensatz stammen. Der Generator wird trainiert, um den Diskriminator zu täuschen, während der Diskriminator trainiert wird, um den Generator zu entlarven.

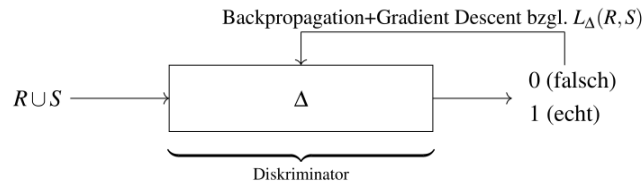


Abbildung 7: Lernschritt eines Diskriminators

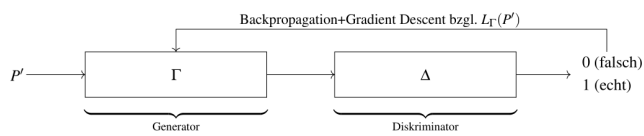


Abbildung 8: Lernschritt eines Generators

Wichtig: Beim Lernschritt des Generators durchläuft der Backpropagation-Algorithmus auch den Diskriminator, passt dort jedoch keine Gewichte an.