Inhalte der Vorlesung

- **Mathematischer Refresher**: Lineare Algebra, Differentiation, Notation
- **Grundlagen Neuronaler Netze**: Perceptron, Universal Approximation Theorem
- **Mathematische Theorie**: Warum funktionieren neuronale Netze?
- **Training und Optimierung**: Gradientenabstieg, Backpropagation, ADAM
- **Deep Learning**: Vanishing Gradients, Aktivierungsfunktionen, tiefe Netze
- **Spezielle Architekturen**: CNNs, RNNs, LSTMs, Autoencoders
- **Predictive Maintenance**: Anwendung auf reale Probleme
- **Praktisches Projekt**: Bearbeitung von Benchmark-Datensätzen

Ziele der Vorlesung - Welche Fragen sollen beantwortet werden?

- **Mathematisch**: Wie funktionieren neuronale Netze wirklich?
- **Theoretisch**: Warum können sie jede Funktion approximieren?
- **Praktisch**: Wie trainiert man sie effizient?
- **Architektur**: Welche speziellen Netze für welche Probleme?
- **Anwendung**: Wann ist Deep Learning die richtige Wahl?
- **Praxis**: Wie löst man reale Probleme mit neuronalen Netzen?

[https://xkcd.com/2451/]

Mathematischer Refresher – Vektoren und Matrizen

- **Vektor**: Spaltenvektor $\mathbf{x} \in \mathbb{R}^d$ mit d Komponenten

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{pmatrix} \tag{1}$$

- **Zeilenvektor**: $\mathbf{x}^T = (x_1, x_2, \ldots, x_d)$ (Transponiert)
- **Matrix**: $\mathbf{A} \in \mathbb{R}^{m \times n}$ mit m Zeilen und n Spalten

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \tag{2}$$

Mathematischer Refresher – Grundoperationen

- **Skalarprodukt** (Dot Product): Für $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$

$$\mathbf{x}^T\mathbf{y} = \mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{d} x_i y_i \tag{3}$$

- **Matrix-Vektor-Multiplikation**: $\mathbf{A} \in \mathbb{R}^{m \times n}, \mathbf{x} \in \mathbb{R}^n$

$$\mathbf{y} = \mathbf{A}\mathbf{x} \in \mathbb{R}^m, \quad y_i = \sum_{j=1}^{n} a_{ij} x_j \tag{4}$$

- **Matrix-Matrix-Multiplikation**: $\mathbf{A} \in \mathbb{R}^{m \times k}, \mathbf{B} \in \mathbb{R}^{k \times n}$

$$\mathbf{C} = \mathbf{A}\mathbf{B} \in \mathbb{R}^{m \times n}, \quad c_{ij} = \sum_{\ell=1}^{k} a_{i\ell} b_{\ell j} \tag{5}$$

- **Elementweise Operationen**: Hadamard-Produkt $\mathbf{A} \odot \mathbf{B}$

$$(\mathbf{A} \odot \mathbf{B})_{ij} = a_{ij} \cdot b_{ij} \tag{6}$$

Mathematischer Refresher – Normen und Abstände

- **Euklidische Norm**: $||\mathbf{x}||_2 = \sqrt{\sum_{i=1}^{d} x_i^2}$ (Länge des Vektors)
- **L1-Norm**: $||\mathbf{x}||_1 = \sum_{i=1}^{d} |x_i|$ (Manhattan-Distanz)
- **Unendlich-Norm**: $||\mathbf{x}||_\infty = \max_i |x_i|$ (Maximum-Norm)
- **Frobenius-Norm** (für Matrizen): $||\mathbf{A}||_F = \sqrt{\sum_{i,j} a_{ij}^2}$
- **Einheitsvektor**: $\mathbf{u}$ mit $||\mathbf{u}||_2 = 1$
- **Orthogonale Vektoren**: $\mathbf{x} \perp \mathbf{y}$ wenn $\mathbf{x}^T\mathbf{y} = 0$
- **Linearkombination**: $\mathbf{y} = \alpha_1\mathbf{x}_1 + \alpha_2\mathbf{x}_2 + \ldots + \alpha_k\mathbf{x}_k$

Mathematischer Refresher – Differentiation und Gradienten

- **Partielle Ableitung**: Für $f : \mathbb{R}^n \to \mathbb{R}$

$$\frac{\partial f}{\partial x_i} = \lim_{h \to 0} \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h} \tag{7}$$

- **Gradient**: Vektor aller partiellen Ableitungen

$$\nabla f(\mathbf{x}) = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \\ \\ \frac{\partial f}{\partial x_n} \end{pmatrix} \tag{8}$$

- **Kettenregel**: Für $f(g(x))$

$$\frac{d}{dx} f(g(x)) = f'(g(x)) \cdot g'(x) \tag{9}$$

- **Multivariable Kettenregel**: Für $f(\mathbf{u}(\mathbf{x}))$

$$\frac{\partial f}{\partial x_i} = \sum_j \frac{\partial f}{\partial u_j} \frac{\partial u_j}{\partial x_i} = \nabla_{\mathbf{u}} f \cdot \frac{\partial \mathbf{u}}{\partial x_i} \tag{10}$$

Mathematischer Refresher – Wichtige Funktionen und Eigenschaften

- **Exponentialfunktion**: e^x, Ableitung: $\frac{d}{dx}e^x = e^x$
- **Logarithmus**: $\ln(x)$, Ableitung: $\frac{d}{dx}\ln(x) = \frac{1}{x}$
- **Sigmoid-Funktion**: $\sigma(x) = \frac{1}{1+e^{-x}}$

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)) \tag{11}$$

- **Quadratische Funktion**: $f(x) = ax^2 + bx + c$, Ableitung: $f'(x) = 2ax + b$
- **Produktregel**: $(fg)' = f'g + fg'$
- **Summenregel**: $(f + g)' = f' + g'$
- **Konstante Faktoren**: $(cf)' = cf'$ für Konstante c

Mathematische Notation – Überblick für diese Vorlesung

- **Skalare**: Kleinbuchstaben a, b, c, x, y, z
- **Vektoren**: Fettgedruckte Kleinbuchstaben $\mathbf{x}, \mathbf{y}, \mathbf{w}, \mathbf{b}$
- **Matrizen**: Fettgedruckte Großbuchstaben $\mathbf{A}, \mathbf{W}, \mathbf{X}$
- **Mengen**: Kalligrafische Buchstaben $\mathcal{D}, \mathcal{M}, \mathcal{X}$
- **Funktionen**: f, g, h, L (Verlustfunktion)
- **Aktivierungsfunktionen**: $\sigma, \text{ReLU}, \tanh$
- **Indizes**:
  - i, j, k: Datenindizes, Neuron-Indizes
  - (ℓ): Schicht-Index, z.B. $\mathbf{W}^{(\ell)}$
  - (t): Zeit-/Iterationsindex, z.B. $\mathbf{w}^{(t)}$
- **Wahrscheinlichkeiten**: $P, p, \mathbb{E}[\cdot]$ (Erwartungswert)
- **Approximation**: $\approx$, Proportionalität: $\propto$

Was sind Neuronale Netze und was ist Deep Learning?

- Abstrakt: Verkettung nichtlinearer Abbildungen
- Die Parameter dieser Abbildungen wird mit vorhandenen Daten ”gelernt”
- Verschiedene Optimierungsverfahren zur Festlegung der ”besten” Parameter

[https://xkcd.com/1838/]

Warum funktionieren Neuronale Netze? – Mathematische Intuition

- **Grundproblem**: Finde eine Funktion $f : \mathbb{R}^d \to \mathbb{R}^k$, die Eingaben $\mathbf{x}$ auf gewünschte Ausgaben $\mathbf{y}$ abbildet
- **Funktionsapproximation**: Neuronale Netze sind universelle Funktionsapproximatoren
- **Komposition einfacher Funktionen**:

$$f(\mathbf{x}) = f_L \circ f_{L-1} \circ \dots \circ f_2 \circ f_1(\mathbf{x}) \tag{12}$$

- Jede Schicht f_i führt eine **affine Transformation** gefolgt von **Nichtlinearität** aus:

$$f_i(\mathbf{x}) = \sigma_i(\mathbf{W}_i\mathbf{x} + \mathbf{b}_i) \tag{13}$$

- **Warum Nichtlinearität wichtig ist**: Ohne sie wäre das gesamte Netz nur eine lineare Transformation

$$\mathbf{W}_L(\mathbf{W}_{L-1}(\dots(\mathbf{W}_1\mathbf{x}))) = (\mathbf{W}_L\mathbf{W}_{L-1}\dots\mathbf{W}_1)\mathbf{x} = \mathbf{W}_{\text{eff}}\mathbf{x} \tag{14}$$

Warum funktionieren Neuronale Netze? – Universal Approximation Theorem

- **Universal Approximation Theorem** [1, 2]:
- Ein Feedforward-Netz mit einer versteckten Schicht kann jede stetige Funktion auf einem kompakten Definitionsbereich beliebig genau approximieren
- **Mathematische Formulierung**: Sei $\sigma : \mathbb{R} \to \mathbb{R}$ eine nicht-konstante, beschränkte und monotone Aktivierungsfunktion. Dann kann für jede stetige Funktion $g : [0, 1]^d \to \mathbb{R}$ und $\epsilon > 0$ eine Funktion

$$F(\mathbf{x}) = \sum_{i=1}^{N} \alpha_i \sigma(\mathbf{w}_i^T \mathbf{x} + b_i) \tag{15}$$

gefunden werden, sodass $|F(\mathbf{x}) - g(\mathbf{x})| < \epsilon$ für alle $\mathbf{x} \in [0, 1]^d$

- **Praktische Bedeutung**:
  - Theoretisch können neuronale Netze jede Funktion lernen
  - Problem: Anzahl der benötigten Neuronen kann exponentiell wachsen
  - Deep Learning: Mehr Schichten können effizienter sein als breitere Netze

Die Mathematik des Lernens – Warum Gradientenabstieg funktioniert

- **Optimierungsproblem**: Minimiere Verlustfunktion $L(\theta)$ über Parameter θ
- **Gradientenabstieg basiert auf Taylor-Entwicklung**:

$$L(\theta + \Delta\theta) \approx L(\theta) + \nabla L(\theta)^T \Delta\theta \tag{16}$$

- Um L zu minimieren, wähle $\Delta\theta = -\eta \nabla L(\theta)$ (mit $\eta > 0$)
- **Warum funktioniert das?** Für kleine η:

$$L(\theta - \eta \nabla L(\theta)) \approx L(\theta) - \eta ||\nabla L(\theta)||^2 \leq L(\theta) \tag{17}$$

- **Konvergenz-Eigenschaften**:
  - Für konvexe Funktionen: Garantierte Konvergenz zum globalen Minimum
  - Für nicht-konvexe Funktionen (neuronale Netze): Konvergenz zu lokalen Minima
  - **Überraschung**: Lokale Minima sind oft ”gut genug” für praktische Anwendungen

Konstruktion von Neuronalen Netzen: Single-Layer-Perceptron

- Einfacher binärer Klassifikator mit Aktivierungsfunktion

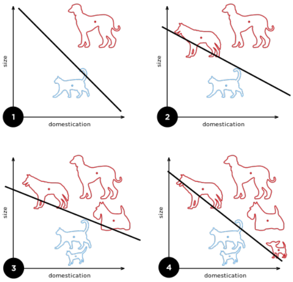
$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w}^T\mathbf{x} + b \geq \theta, \\ 0 & \text{otherwise} \end{cases} \tag{18}$$

- mit dem Gewichtsvektor $\mathbf{w} \in \mathbb{R}^d$, Eingabevektor $\mathbf{x} \in \mathbb{R}^d$, Bias $b \in \mathbb{R}$ und Schwellwert θ
- Das Skalarprodukt: $\mathbf{w}^T\mathbf{x} = \sum_{i=1}^{d} w_i x_i$
- Entscheidungsgrenze im 2D-Fall ($d = 2$): Gerade mit Gleichung

$$w_1x_1 + w_2x_2 + b = \theta \tag{19}$$

$$x_2 = -\frac{w_1}{w_2}x_1 - \frac{b-\theta}{w_2} \tag{20}$$

- Geometrische Interpretation: Hyperebene teilt den $\mathbb{R}^d$ in zwei Halbräume
- Linear separierbare Probleme: Klassen können durch Hyperebene getrennt werden



Training von Neuronalen Netzen: Single-Layer-Perceptron

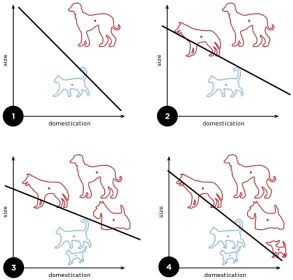
- Gegeben: Trainingsdatensatz $D = \{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$ mit $\mathbf{x}_i \in \mathbb{R}^d, y_i \in \{-1, +1\}$
- Perceptron-Lernregel [3]:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \eta \sum_{(\mathbf{x}_i, y_i) \in M^{(t)}} y_i \mathbf{x}_i \tag{21}$$

- Fehlklassifizierungen: $M^{(t)} = \{(\mathbf{x}_i, y_i) : y_i(\mathbf{w}^{(t)T}\mathbf{x}_i + b) \leq 0\}$
- Verlustfunktion (Perceptron-Verlust):

$$L(\mathbf{w}, b) = \sum_{(\mathbf{x}_i, y_i) \in M} -y_i(\mathbf{w}^T\mathbf{x}_i + b) \tag{22}$$

- Konvergenz-Theorem: Für linear separierbare Daten konvergiert der Algorithmus in endlich vielen Schritten
- Margin $\gamma = \min_i \frac{y_i(\mathbf{w}^{*T}\mathbf{x}_i + b^*)}{||\mathbf{w}^*||_2}$ bestimmt Konvergenzgeschwindigkeit



Training von Neuronalen Netzen: Gradientenabstieg

- **Gradientenabstieg** (Gradient Descent) - allgemeines Optimierungsverfahren
- Iterative Aktualisierung der Parameter:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta L(\theta^{(t)}) \tag{23}$$

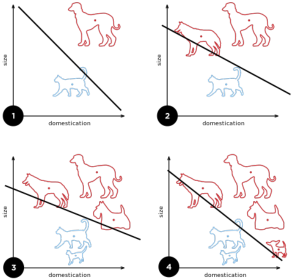
- $\eta > 0$: Lernrate (Schrittweite), θ: Parametervektor
- Gradient einer Funktion $f : \mathbb{R}^n \to \mathbb{R}$:

$$\nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_n} \right)^T \tag{24}$$

- Gradient zeigt in Richtung des steilsten Anstiegs $\Rightarrow -\nabla f$ zeigt zum lokalen Minimum
- Für Perceptron-Verlust:

$$\frac{\partial L}{\partial w_j} = \sum_{(\mathbf{x}_i, y_i) \in M} -y_i x_{ij} \tag{25}$$

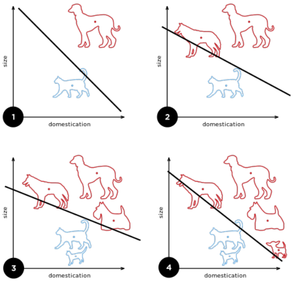
$$\nabla_\mathbf{w} L = - \sum_{(\mathbf{x}_i, y_i) \in M} y_i \mathbf{x}_i \tag{26}$$



Training von Neuronalen Netzen: Single-Layer-Perceptron

- Daraus folgt:

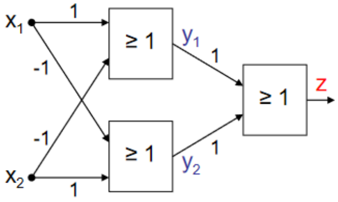
$$\nabla f(w) = \left(- \sum_{x \in F(w)} x_1, - \sum_{x \in F(w)} x_2, \ldots, - \sum_{x \in F(w)} x_n \right) = - \sum_{x \in F(w)} x \qquad (27)$$



Grenzen des Single-Layer-Perceptrons – Das XOR-Problem

- **Fundamentale Limitation**: Perceptron kann nur linear separierbare Probleme lösen
- **XOR-Problem** [4]:

x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0



- **Mathematischer Beweis der Unmöglichkeit**:
- Angenommen, es existiert $\mathbf{w} = (w_1, w_2)$ und b, sodass:

$$w_1 \cdot 0 + w_2 \cdot 1 + b > 0 \quad \text{(für (0,1))} \tag{28}$$

$$w_1 \cdot 1 + w_2 \cdot 0 + b > 0 \quad \text{(für (1,0))} \tag{29}$$

$$w_1 \cdot 0 + w_2 \cdot 0 + b \leq 0 \quad \text{(für (0,0))} \tag{30}$$

$$w_1 \cdot 1 + w_2 \cdot 1 + b \leq 0 \quad \text{(für (1,1))} \tag{31}$$

- Aus (1) und (3): $w_2 > -b \geq 0 \Rightarrow w_2 > 0$

Konstruktion von Neuronalen Netzen: Multi-Layer-Perceptron

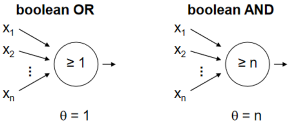
- **Lösung des XOR-Problems**: Mehrschichtige Netze!
- **Komposition von Hyperebenen**:
  - Erste Schicht: Erzeugt mehrere lineare Entscheidungsgrenzen
  - Zweite Schicht: Kombiniert diese zu komplexeren Formen
- **Mathematische Intuition für XOR**:

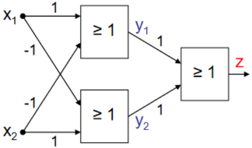
$$h_1 = \sigma(x_1 + x_2 - 0.5) \quad \text{(OR-Gate)} \tag{32}$$

$$h_2 = \sigma(-x_1 - x_2 + 1.5) \quad \text{(NAND-Gate)} \tag{33}$$

$$\text{XOR} = \sigma(h_1 + h_2 - 1.5) \tag{34}$$

- **Universal Approximation**: Mit einer versteckten Schicht können beliebige stetige Funktionen approximiert werden
- **Tiefe vs. Breite**: Tiefere Netze können effizienter sein als breitere





Training von Neuronalen Netzen: Multi-Layer-Perceptron

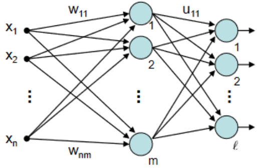
- **Multilayer Perceptron (MLP)**: Neuronales Netz mit versteckten Schichten
- Forward-Pass für 2-Schicht-Netz:

$$\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \quad \text{(lineare Transformation)} \tag{35}$$

$$\mathbf{a}^{(1)} = \sigma(\mathbf{z}^{(1)}) \quad \text{(Aktivierung)} \tag{36}$$

$$\mathbf{z}^{(2)} = \mathbf{W}^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)} \tag{37}$$

$$\hat{\mathbf{y}} = \sigma(\mathbf{z}^{(2)}) \tag{38}$$



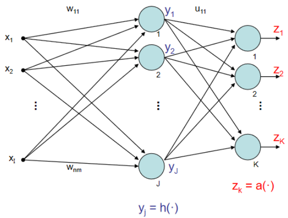
- Mean Squared Error (MSE) Loss:

$$L(\mathbf{W}, \mathbf{b}) = \frac{1}{n}\sum_{i=1}^{n} ||\hat{\mathbf{y}}_i - \mathbf{y}_i||_2^2 \tag{39}$$

- Problem der Heaviside-Funktion: $H(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$ nicht differenzierbar
- Lösung: Glatte Aktivierungsfunktionen (Sigmoid, Tanh, ReLU)

Training von Neuronalen Netzen: Multi-Layer-Perceptron

- $$f(w) = \sum_{x \in B} ||g(w;x) - g^*(x)||^2 \quad \to \min \tag{40}$$
- mit dem Output des Netzes $g(w;x)$ und dem erwarteten Output $g^*(x)$
- $$u^{(t+1)} = u^t - \gamma \nabla_u f(w_t, u_t) \tag{41}$$
  $$w^{(t+1)} = w^t - \gamma \nabla_w f(w_t, u_t) \tag{42}$$
  (43)



- x_i: Inputs
- y_j: Werte nach dem ersten Layer
- z_k: Werte nach dem zweiten Layer

Backpropagation-Algorithmus: Mathematische Grundlagen

- **Backpropagation** [5]: Effizienter Algorithmus zur Berechnung von Gradienten in neuronalen Netzen
- Anwendung der Kettenregel der Differentiation:

$$\frac{\partial L}{\partial w_{ij}^{(l)}} = \frac{\partial L}{\partial z_j^{(l)}} \cdot \frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}} \tag{44}$$

- Definition der **lokalen Gradienten** (Deltas):

$$\delta_j^{(l)} = \frac{\partial L}{\partial z_j^{(l)}} \tag{45}$$

- Rekursive Berechnung (rückwärts durch das Netz):

$$\delta_j^{(L)} = \frac{\partial L}{\partial a_j^{(L)}} \cdot \sigma'(z_j^{(L)}) \quad \text{(Output-Schicht)} \tag{46}$$

$$\delta_j^{(l)} = \left(\sum_k \delta_k^{(l+1)} w_{jk}^{(l+1)}\right) \sigma'(z_j^{(l)}) \quad \text{(versteckte Schichten)} \tag{47}$$

- Gradientenberechnung:

$$\frac{\partial L}{} = \delta_j^{(l)} \cdot a_i^{(l-1)} \tag{48}$$

Training von Neuronalen Netzen: Multi-Layer-Perceptron

- analog zum SLP nutzen wir den Gradienten zur Minimierung des Fehlers

$$\nabla f(w,u) = \sum_{x,z^* \in B} \nabla f(w,u;x,z^*)$$

$$\frac{\partial f(w,u)}{\partial u_{jk}} = \sum_{x,z^* \in B} \frac{\partial f(w,u;x,z^*)}{\partial u_{jk}}$$

$$\frac{\partial f(w,u)}{\partial w_{ij}} = \sum_{x,z^* \in B} \frac{\partial f(w,u;x,z^*)}{\partial w_{ij}}$$

- mit den Sigmoid-Aktivierungsfunktionen:

$$a(x) = h(x) = \frac{1}{1+e^{-x}} \text{ mit der Ableitung: } \frac{\mathrm{d}a(x)}{\mathrm{d}x} = a(x) \cdot (1 - a(x))$$

- mit der Kettenregel für Ableitungen:

$$[p(q(x))]' = p'(q(x)) \cdot q'(x)$$

- ergibt sich für den Gradienten von f:

$$f(w,u;x,z^*) = \sum_{k=1}^{K} [a(u_k y) - z_k^*]^2$$

Training von Neuronalen Netzen: Multi-Layer-Perceptron

- ergibt sich für den Gradienten von f:

$$f(w,u;x,z^*) = \sum_{k=1}^{K} [a(u_k y) - z_k^*]^2$$

$$\frac{\partial f(w,u;x,z^*)}{\partial u_{jk}} = \sum_{x,z^* \in B} \frac{\partial f(w,u;x,z^*)}{\partial u_{jk}}$$

$$= 2[a(u'_k y) - z_k^*] \cdot a'(u'_k y) \cdot y_j$$

$$= 2[a(u'_k y) - z_k^*] \cdot a(u'_k y \cdot (1 - a(u'_k y))) \cdot y_j$$

$$= \underbrace{2[z_k - z_k^*] \cdot z_k \cdot (1 - z_k)}_{\text{Fehlerterm } \delta_k} \cdot y_j$$

$$\frac{\partial f(w,u;x,z^*)}{\partial w_{ij}} = 2 \sum_{k=1}^{K} (a(u'_k y) - z_k^*) \cdot a'(u'_k y) \cdot u_{jk} \cdot h'(w'_j x) x_i$$

$$= 2 \sum_{k=1}^{K} (z_k - z_k^*) \cdot z_k \cdot (1 - z_k) \cdot u_{jk} \cdot y_j (1 - y_j) x_i$$

$$= x_i y_j (1 - y_j) \underbrace{2 \sum_{k=1}^{K} (z_k - z_k^*) \cdot z_k \cdot (1 - z_k) \cdot u_{jk}} \cdot y_j$$

Verallgemeinerung Training von Neuronalen Netzen: M-Layer-Perceptron

- bei einem Neuronalen Netz mi L Layern $S_1, S_2, ..., S_L$
- den Gewichten w_{ij} in der Matrix W
- dem output eines Neurons o_j
- ist der Fehlerterm:

$$\delta_j = \begin{cases} o_j \cdot (1 - o_j) \cdot (o_j - z_j*) & \text{if } j \in S_L \text{, output Neuron} \\ o_j \cdot (1 - o_j) \cdot \sum_{k \in S_{m+1}} \delta_k \cdot w_{jk} & \text{if } j \in S_m \text{and } m < L \end{cases}$$

- Der Korrekturterm für die einzelnen Gewichte ist dann:

$$w_{ij}^{(t+1)} = w_{ij}^{t} - \gamma \cdot o_i \cdot \delta_j \tag{50}$$

Fortgeschrittene Optimierungsalgorithmen

- **Momentum**: Beschleunigung in konsistente Richtungen

$$\mathbf{v}^{(t+1)} = \beta \mathbf{v}^{(t)} + \eta \nabla L(\theta^{(t)}) \quad (51)$$

$$\theta^{(t+1)} = \theta^{(t)} - \mathbf{v}^{(t+1)} \quad (52)$$

- **ADAM** [6] (Adaptive Moment Estimation) - Kombination aus Momentum und RMSprop:

$$m_w^{(t+1)} = \beta_1 m_w^{(t)} + (1 - \beta_1)\nabla_w L^{(t)} \quad \text{(1. Moment)} \quad (53)$$

$$v_w^{(t+1)} = \beta_2 v_w^{(t)} + (1 - \beta_2)(\nabla_w L^{(t)})^2 \quad \text{(2. Moment)} \quad (54)$$

$$\hat{m}_w^{(t)} = \frac{m_w^{(t+1)}}{1 - \beta_1^{t+1}} \quad \text{(Bias-Korrektur)} \quad (55)$$

$$\hat{v}_w^{(t)} = \frac{v_w^{(t+1)}}{1 - \beta_2^{t+1}} \quad \text{(Bias-Korrektur)} \quad (56)$$

$$w^{(t+1)} = w^{(t)} - \frac{\eta}{\sqrt{\hat{v}_w^{(t)}} + \epsilon} \hat{m}_w^{(t)} \quad (57)$$

- Typische Hyperparameter: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$

Warum "Deep" Learning? – Mathematische Rechtfertigung für tiefe Netze

- **Representation Learning**: Tiefere Netze lernen hierarchische Merkmalsdarstellungen
- **Mathematischer Vorteil**: Exponentiell weniger Parameter für dieselbe Expressivität
- **Beispiel - Parität-Funktion**: Prüfe, ob eine ungerade Anzahl von Bits gesetzt ist
  - Flaches Netz: Benötigt $O(2^n)$ versteckte Neuronen
  - Tiefes Netz: Benötigt nur $O(n)$ Neuronen in $O(\log n)$ Schichten
- **Kompositionelle Struktur**: Viele reale Funktionen haben hierarchische Struktur

$$f(\mathbf{x}) = g_L(g_{L-1}(\ldots g_2(g_1(\mathbf{x}))\ldots)) \tag{58}$$

- **Feature Learning**: Jede Schicht ℓ lernt Features der Form:

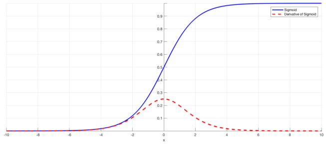
$$\mathbf{h}^{(\ell)} = \sigma(\mathbf{W}^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}) \tag{59}$$

- **Intuition**:
  - Untere Schichten: Einfache Features (Kanten, Texturen)
  - Mittlere Schichten: Kombinationen (Formen, Teile)
  - Obere Schichten: Komplexe Konzepte (Objekte, Semantik)

Das Vanishing Gradient Problem – Warum tiefe Netze schwer zu trainieren sind

- **Problem**: Bei tiefen Netzen werden Gradienten exponentiell kleiner
- **Mathematische Analyse**: Für Sigmoid-Aktivierung $\sigma'(x) \leq 0.25$
- Gradient in Schicht ℓ proportional zu:

$$\frac{\partial L}{\partial \mathbf{W}^{(\ell)}} \propto \prod_{i=\ell+1}^{L} \mathbf{W}^{(i)} \sigma'(\mathbf{z}^{(i)}) \tag{60}$$



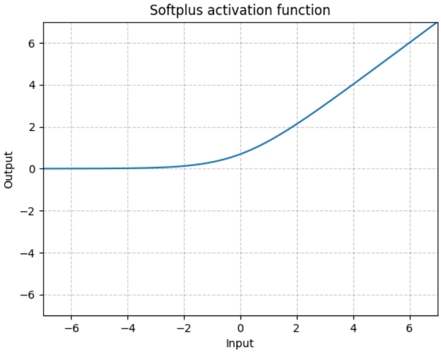
- Für $L - \ell$ Schichten: Faktor $\leq (0.25)^{L-\ell}$
- **Beispiel**: Bei 10 Schichten kann Gradient um Faktor 10^{-6} schrumpfen!
- **Lösungsansätze**:
  - ReLU-Aktivierungen: $\text{ReLU}'(x) = 1$ für $x > 0$
  - Residual Connections (ResNets)
  - Normalization (BatchNorm, LayerNorm)
  - Bessere Initialisierung (Xavier, He)

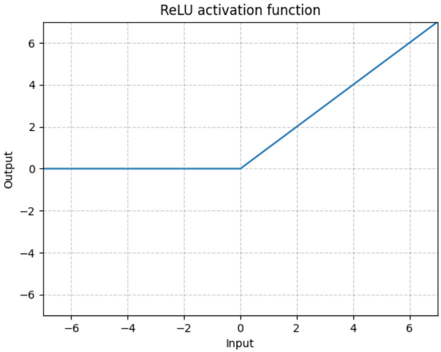
Aktivierungsfunktionen – Mathematische Eigenschaften und Warum sie wichtig sind

- **Sigmoid-Funktion**: $\sigma(x) = \frac{1}{1+e^{-x}}$, Ableitung: $\sigma'(x) = \sigma(x)(1 - \sigma(x))$
  - Glatt und differenzierbar, Ausgabe in $(0, 1)$
  - **Problem**: Vanishing Gradients für $|x| >> 0$: $\sigma'(x) \to 0$
- **Tanh-Funktion**: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$, Ableitung: $\tanh'(x) = 1 - \tanh^2(x)$
  - Ausgabe in $(-1, 1)$, zero-centered (bessere Konvergenz)
  - Ebenfalls Vanishing Gradient Problem
- **ReLU-Funktion** [7]: $\text{ReLU}(x) = \max(0, x)$, Ableitung: $\text{ReLU}'(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$
  - Löst Vanishing Gradient Problem für $x > 0$
  - Computationally efficient, führt zu sparse representations
  - **Problem**: "Dying ReLU" - Neuronen können "sterben" wenn $x \leq 0$
- **Warum Nichtlinearität essentiell ist**:

$$\text{Ohne } \sigma : f(\mathbf{x}) = \mathbf{W}_2(\mathbf{W}_1\mathbf{x}) = (\mathbf{W}_2\mathbf{W}_1)\mathbf{x} = \mathbf{W}_{\text{linear}}\mathbf{x} \quad (61)$$

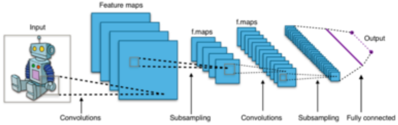
Häufige Aktivierungsfunktionen – Visualisierung

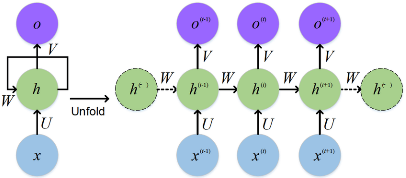




Weitere Netzwerk Architekturen

- spezielle Datenstrukturen profitieren von speziellen Architekturen
- Bilderkennung →Convolutional Neural Network (CNN)
- sequentielle Daten →Recurrent Neural Networks (RNN)
- im speziellen um Kausalität/Kontext herzustellen →Long Short Term Memory (LSTM)





Warum funktionieren CNNs? – Mathematische Prinzipien

- **Drei Schlüsselprinzipien von CNNs**:
- **1. Lokale Konnektivität**: Jedes Neuron ist nur mit lokalem Bereich verbunden

$$y_{i,j}^{(\ell)} = \sigma\left(\sum_{m=-k}^{k}\sum_{n=-k}^{k} w_{m,n}^{(\ell)} \cdot x_{i+m,j+n}^{(\ell-1)} + b^{(\ell)}\right) \tag{62}$$

- **2. Parameter Sharing**: Derselbe Filter $\mathbf{W}$ wird über gesamte Feature-Map verwendet
  - Reduziert Parameter von $O(H \cdot W \cdot d^2)$ auf $O(k^2 \cdot d)$
  - Erzwingt Translationsinvarianz
- **3. Equivarianz zu Translationen**: Wenn Input um $\mathbf{t}$ verschoben wird, verschiebt sich Output ebenfalls um $\mathbf{t}$

$$\mathrm{Conv}(T_{\mathbf{t}}[\mathbf{x}]) = T_{\mathbf{t}}[\mathrm{Conv}(\mathbf{x})] \tag{63}$$

- **Pooling** führt zu begrenzter Translationsinvarianz:

$$\mathrm{MaxPool}(\mathbf{X})_{i,j} = \max_{(p,q)\in N_{i,j}} \mathbf{X}_{p,q} \tag{64}$$

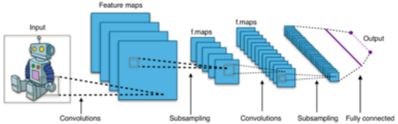
- **Hierarchische Feature-Extraktion**: Einfache → Komplexe Features

Spezielle Netzwerkarchitekturen: CNN – Implementierung

- **Convolutional Neural Networks** [8]: Spezialisiert auf gitterförmige Daten (Bilder)

- **Faltungsoperation** (2D-Convolution):

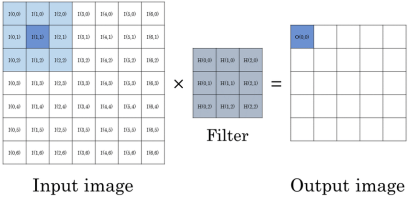
$$(\mathbf{I} * \mathbf{K})_{i,j} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \mathbf{I}_{i+m,j+n} \cdot \mathbf{K}_{m,n} \qquad (65)$$



- $\mathbf{I}$: Input-Feature-Map, $\mathbf{K}$: Kernel (Filter) der Größe $M \times N$

- **Pooling**: Dimensionsreduktion, z.B. Max-Pooling:

$$\text{MaxPool}(\mathbf{X})_{i,j} = \max_{p,q \in P_{i,j}} \mathbf{X}_{p,q} \qquad (66)$$

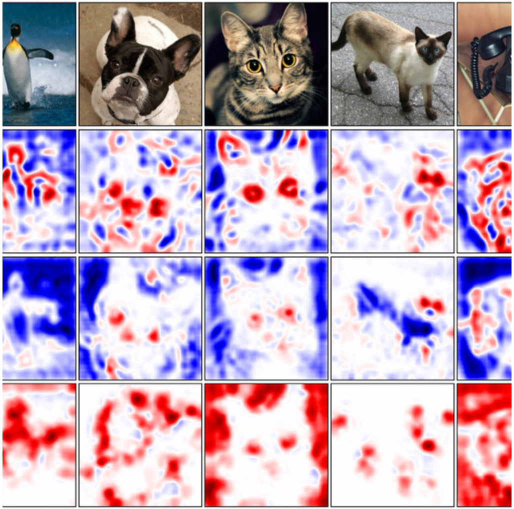


- **Parameter Sharing**: Derselbe Filter wird über gesamte Feature-Map angewendet

- **Translation Invariance**: Robustheit gegenüber Verschiebungen

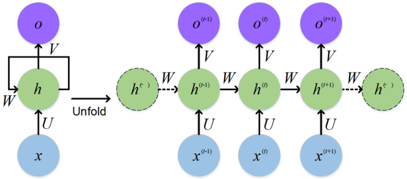
Spezielle Netzwerkarchitekturen: CNN

- Anschaulich: Formen werden erkannt
- Katzenohren sind anders als Hundeohren
- Verallgemeinerbar für andere Objektklassifizierungen
- für Details die XAI Vorlesung nächstes Semester



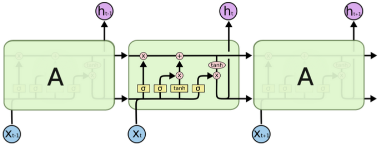
Spezielle Netzwerkarchitekturen: RNN

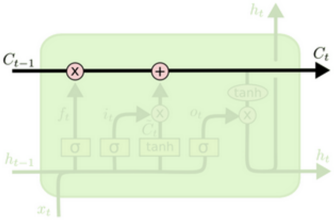
- Anschaulich: Schleife in Netzwerk ”merkt” sich voherige Zustände
- funktioniert für kurze Zeiträume
- Entfaltung eines RNN →viele zu trainierende Gewichte
- Problem: langfristige Zusammenhänge werden nicht erfasst
- Problem: Verschwindende Gradienten



Spezielle Netzwerkarchitekturen: LSTM

- **Long Short-Term Memory** [9]: Lösung des Vanishing Gradient Problems in RNNs
- **Cell State $\mathbf{C}_t$**: Langzeit-Gedächtnis der LSTM-Zelle
- **Hidden State $\mathbf{h}_t$**: Kurzzeit-Output der Zelle
- Drei Gating-Mechanismen kontrollieren Informationsfluss:
  - Forget Gate: Welche Informationen vergessen?
  - Input Gate: Welche neuen Informationen speichern?
  - Output Gate: Welche Teile des Cell States ausgeben?





LSTM: Mathematische Formulierung

- **Forget Gate**: Entscheidet, welche Informationen aus $\mathbf{C}_{t-1}$ gelöscht werden

$$\mathbf{f}_t = \sigma(\mathbf{W}_f \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f) \tag{67}$$

- **Input Gate**: Bestimmt neue Informationen für Cell State

$$\mathbf{i}_t = \sigma(\mathbf{W}_i \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i) \tag{68}$$

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{W}_C \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_C) \tag{69}$$

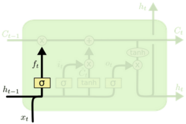
- **Cell State Update**:

$$\mathbf{C}_t = \mathbf{f}_t * \mathbf{C}_{t-1} + \mathbf{i}_t * \tilde{\mathbf{C}}_t \tag{70}$$

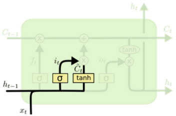
- **Output Gate** und **Hidden State**:

$$\mathbf{o}_t = \sigma(\mathbf{W}_o \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o) \tag{71}$$

$$\mathbf{h}_t = \mathbf{o}_t * \tanh(\mathbf{C}_t) \tag{72}$$



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

LSTM: Cell State Update und Output

- **Cell State Update**: Kombination von alten und neuen Informationen

$$\mathbf{C}_t = \mathbf{f}_t \odot \mathbf{C}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{C}}_t \qquad (73)$$

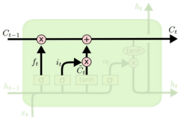
- **Output Gate**: Bestimmt, welche Teile des Cell States ausgegeben werden

$$\mathbf{o}_t = \sigma(\mathbf{W}_o \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o) \qquad (74)$$

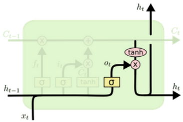
- **Hidden State**: Gefilterte Version des Cell States

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{C}_t) \qquad (75)$$

- **Warum funktioniert LSTM?**
  - Cell State kann Informationen über viele Zeitschritte transportieren
  - Gates kontrollieren selektiv Informationsfluss
  - Löst das Vanishing Gradient Problem von Standard-RNNs



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$



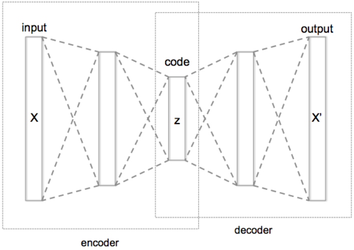
$$o_t = \sigma\left(W_o \; [h_{t-1}, x_t] \; + \; b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Autoencoder: Grundlagen und Mathematik

- **Autoencoder**: Unsupervised Learning zur Dimensionsreduktion und Rekonstruktion
- **Encoder**: $\mathbf{z} = f_{enc}(\mathbf{x}; \theta_{enc})$ - Komprimierung in latenten Raum
- **Decoder**: $\hat{\mathbf{x}} = f_{dec}(\mathbf{z}; \theta_{dec})$ - Rekonstruktion aus latenter Repräsentation
- **Verlustfunktion**: Rekonstruktionsfehler

$$L(\mathbf{x}, \hat{\mathbf{x}}) = ||\mathbf{x} - \hat{\mathbf{x}}||^2 \tag{76}$$

- **Latenter Raum $\mathbf{z} \in \mathbb{R}^d$** mit d << Eingabedimension
- **Anwendungen**:
  - Dimensionsreduktion (wie PCA, aber nichtlinear)
  - Anomaly Detection (hoher Rekonstruktionsfehler)
  - Denoising (rauschhafte Eingaben, saubere Ziele)
  - Feature Learning für Downstream-Tasks



Variational Autoencoders (VAEs): Motivation

- **Problem klassischer Autoencoders**: Latenter Raum ist nicht interpretierbar oder strukturiert
- **Ziel der VAEs**: Lernen einer **probabilistischen** latenten Repräsentation
- **Generative Modellierung**: $p(\mathbf{x}) = \int p(\mathbf{x}|\mathbf{z})p(\mathbf{z})d\mathbf{z}$
- **Bayessche Perspektive**:
  - Prior: $p(\mathbf{z}) = N(\mathbf{0}, \mathbf{I})$ (Standard-Normalverteilung)
  - Likelihood: $p(\mathbf{x}|\mathbf{z})$ durch Decoder-Netzwerk
  - Posterior: $p(\mathbf{z}|\mathbf{x})$ durch Encoder-Netzwerk approximiert
- **Herausforderung**: $p(\mathbf{z}|\mathbf{x})$ ist analytisch intraktabel
- **Lösung**: Variational Inference mit **Evidence Lower Bound (ELBO)**
- **Vorteil**: Latenter Raum ist **kontinuierlich** und **interpolierbar**
- **Anwendungen**: Bildgenerierung, Data Augmentation, Semi-Supervised Learning

VAEs: Mathematische Grundlagen

- **Variational Inference**: Approximiere $p(\mathbf{z}|\mathbf{x})$ durch $q_\phi(\mathbf{z}|\mathbf{x})$
- **Evidence Lower Bound (ELBO)**:

$$\log p(\mathbf{x}) \geq \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})] - D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p(\mathbf{z})) \tag{77}$$

- **Zwei Terme der Verlustfunktion**:
  - **Rekonstruktionsverlust**: $\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})]$
  - **Regularisierungsterm**: $D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p(\mathbf{z}))$
- **Encoder**: $q_\phi(\mathbf{z}|\mathbf{x}) = N(\mu_\phi(\mathbf{x}), \mathrm{diag}(\sigma^2_\phi(\mathbf{x})))$
- **Decoder**: $p_\theta(\mathbf{x}|\mathbf{z}) = N(\mu_\theta(\mathbf{z}), \sigma^2_\theta \mathbf{I})$
- **KL-Divergenz** (analytisch berechenbar für Gausssche Verteilungen):

$$D_{KL} = \frac{1}{2}\sum_{j=1}^{d}(1 + \log(\sigma_j^2) - \mu_j^2 - \sigma_j^2) \tag{78}$$

VAEs: Reparameterization Trick

- **Problem**: Stochastisches Sampling $\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})$ ist nicht differenzierbar
- **Lösung**: Reparameterization Trick (Kingma & Welling, 2013)
- **Statt**: $\mathbf{z} \sim N(\mu, \sigma^2)$
- **Verwende**:

$$\epsilon \sim N(\mathbf{0}, \mathbf{I}) \quad \text{(deterministisches Rauschen)} \tag{79}$$

$$\mathbf{z} = \mu + \sigma \odot \epsilon \quad \text{(deterministische Transformation)} \tag{80}$$

- **Vorteil**: Gradienten können durch μ und σ zurückpropagiert werden
- **Praktische Implementierung**:
  - Encoder gibt $\mu(\mathbf{x})$ und $\log \sigma^2(\mathbf{x})$ aus
  - Sample $\epsilon \sim N(\mathbf{0}, \mathbf{I})$
  - Berechne $\mathbf{z} = \mu + \exp(\frac{1}{2} \log \sigma^2) \odot \epsilon$
  - Führe $\mathbf{z}$ durch Decoder
- **Trainingsalgorithmus**: Standard-Backpropagation mit stochastischem Gradienten!

VAEs: Eigenschaften und Anwendungen

- **Interpolation im latenten Raum**: Glatte Übergänge zwischen Datenpunkten

$$\mathbf{z}_{interp} = \alpha\mathbf{z}_1 + (1 - \alpha)\mathbf{z}_2, \quad \alpha \in [0, 1] \tag{81}$$

- **Generierung neuer Daten**: Sample $\mathbf{z} \sim N(\mathbf{0}, \mathbf{I})$, dann $\mathbf{x}_{new} = \text{Decoder}(\mathbf{z})$
- **Disentangled Representations**: Verschiedene Dimensionen von $\mathbf{z}$ kodieren verschiedene Eigenschaften
- **Anwendungen**:
  - **Computer Vision**: Gesichtsgenerierung, Style Transfer
  - **NLP**: Text Generation, Sentence Interpolation
  - **Drug Discovery**: Molekularstruktur-Generation
  - **Anomaly Detection**: Niedrige Likelihood → Anomalie
  - **Data Augmentation**: Generierung synthetischer Trainingsdaten
- **Varianten**:
  - β-VAE: $\beta \cdot D_{KL}$ für bessere Disentanglement
  - Conditional VAE: $p(\mathbf{x}|\mathbf{y}, \mathbf{z})$ mit Labels $\mathbf{y}$
  - Hierarchical VAE: Mehrere latente Schichten

GANs vs. VAEs: Vergleich der generativen Modelle

- **Generative Adversarial Networks (GANs)**:
  - Adversarial Training: Generator vs. Discriminator
  - Sehr scharfe, realistische Bilder
  - Training instabil, Mode Collapse möglich
  - Kein Encoder - keine direkte latente Repräsentation von Daten
- **Variational Autoencoders (VAEs)**:
  - Likelihood-basiertes Training mit ELBO
  - Verschwommene Bilder durch Gauss-Annahme
  - Stabiles Training, theoretisch fundiert
  - Bidirektional: Encoding und Decoding möglich
- **Praktische Wahl**:
  - **GANs**: Wenn Bildqualität wichtigster Faktor
  - **VAEs**: Wenn interpretierbare latente Repräsentation wichtig
  - **Hybrid-Modelle**: VAE-GAN kombiniert beide Ansätze

References I

G. Cybenko, “Approximation by superpositions of a sigmoidal function,” Mathematics of control, signals and systems, vol. 2, no. 4, pp. 303–314, 1989.

K. Hornik, “Approximation capabilities of multilayer feedforward networks,” Neural networks, vol. 4, no. 2, pp. 251–257, 1991.

F. Rosenblatt, “The perceptron: a probabilistic model for information storage and organization in the brain,” Psychological review, vol. 65, no. 6, pp. 386–408, 1958.

M. Minsky and S. Papert, Perceptrons: An introduction to computational geometry.
MIT press, 1969.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” Nature, vol. 323, no. 6088, pp. 533–536, 1986.

D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” arXiv preprint arXiv:1412.6980, 2014.

V. Nair and G. E. Hinton, “Rectified linear units improve restricted boltzmann machines,” in Proceedings of the 27th international conference on machine learning (ICML-10), pp. 807–814, 2010.

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” Proceedings of the IEEE, vol. 86, no. 11, pp. 2278–2324, 1998.

References II

S. Hochreiter and J. Schmidhuber, “Long short-term memory,” Neural computation, vol. 9, no. 8, pp. 1735–1780, 1997.