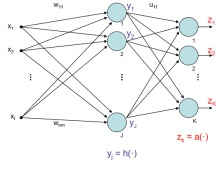


Neuronale Netze/Deep Learning und Predictive Maintenance

Felix Neubürger

08. January 2026

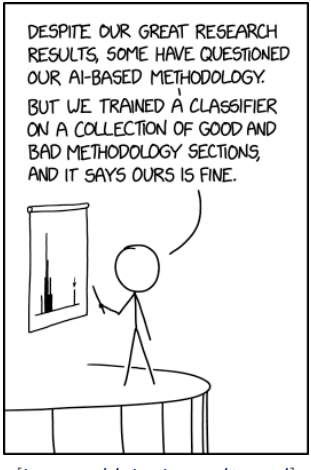
Fachhochschule Südwestfalen, Ingenieurs- & Wirtschaftswissenschaften

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, GANs, Autoencoders

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?



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

Mathematischer Refresher – Vektoren und Matrizen

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

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

- **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

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

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

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

- **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**: $\sigma(x) = \frac{1}{1+e^{-x}}$

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

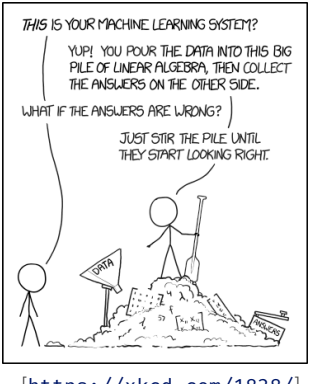
- **Quadratische Funktion**: $f(x) = ax^2 + bx + c$, Ableitung: $f'(x) = 2ax + b$
- **Produktregel**: $(fg)' = f'g + fg'$
- **Summenregel**: $(f + g)' = f' + g'$
- **Konstante Faktoren**: $(cf)' = cf'$ für Konstante c

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 $\mathbf{A}, \mathbf{W}, \mathbf{X}$
- **Funktionen**: f, g, h, L (Verlustfunktion)
- **Aktivierungsfunktionen**: σ, 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**:

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

- 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

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

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:

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

$$\leq L(\theta) \tag{18}$$

- **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

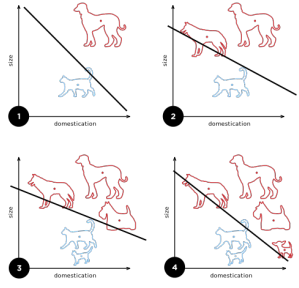
- Einfacher binärer Klassifikator mit Aktivierungsfunktion

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w}^T\mathbf{x} + b \geq \theta, \\ 0 & \text{otherwise} \end{cases} \tag{19}$$

- 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

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

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{b - \theta}{w_2} \tag{21}$$



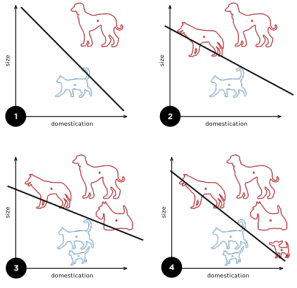
- Geometrische Interpretation: Hyperebene teilt den $\mathbb{R}^d$ in zwei Halbräume
- Linear separierbare Probleme:
  Klassen können durch Hyperebene getrennt werden

Perceptron Training: Lernalgorithmus

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

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \eta \sum_{(\mathbf{x}_i, y_i) \in F^{(t)}} y_i \mathbf{x}_i \tag{22}$$

- **Fehlklassifizierungen**:
  $F^{(t)} = \{(\mathbf{x}_i, y_i) : y_i(\mathbf{w}^{(t)T}\mathbf{x}_i) \leq 0\}$
- **Algorithmus**:
  1. Initialisiere $\mathbf{w}^{(0)} = \mathbf{0}$
  2. Für jede Fehlklassifikation: Update $\mathbf{w}$
  3. Wiederhole bis Konvergenz
- **Konvergenz-Garantie**:
  Für linear separierbare Daten konvergiert in endlich vielen Schritten



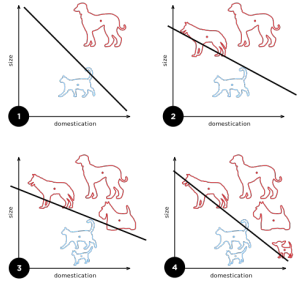
Gradientenabstieg: Allgemeines Optimierungsverfahren

- **Grundprinzip**: Iterative Minimierung der Verlustfunktion
- **Update-Regel**:

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

- **Parameter**:
  - $\eta > 0$: Lernrate (Schrittweite)
  - θ: Parametervektor (Gewichte)
  - ∇L: Gradient der Verlustfunktion
- **Intuition**: Gradient zeigt bergauf → $-\nabla L$ zeigt zum Minimum
- **Perceptron-Gradient**:

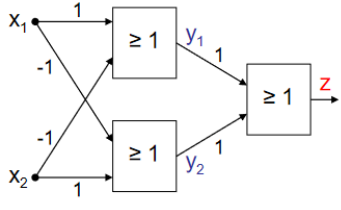
$$\nabla_{\mathbf{w}} L = -\sum_{(\mathbf{x}_i, y_i) \in F} y_i \mathbf{x}_i \tag{24}$$



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:

$$w_1 \cdot 0 + w_2 \cdot 1 + b > 0 \quad \text{(für (0,1))} \tag{25}$$
$$w_1 \cdot 1 + w_2 \cdot 0 + b > 0 \quad \text{(für (1,0))} \tag{26}$$
$$w_1 \cdot 0 + w_2 \cdot 0 + b \leq 0 \quad \text{(für (0,0))} \tag{27}$$
$$w_1 \cdot 1 + w_2 \cdot 1 + b \leq 0 \quad \text{(für (1,1))} \tag{28}$$

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

Konstruktion von Neuronalen Netzen: Multi-Layer-Perceptron

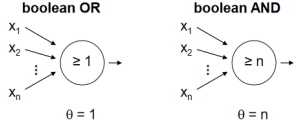
- **Lösung des XOR-Problems**: Mehrschichtige Netze!
- **Komposition von Hyperebenen**:
  - Erste Schicht: Erzeugt mehrere lineare Entscheidungsgrenzen
  - Zweite Schicht: Kombiniert diese zu komplexeren Formen
- **Mathematische Intuition für XOR**:

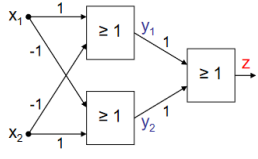
$$h_1 = \sigma(x_1 + x_2 - 0.5) \quad \text{(OR-Gate)} \tag{29}$$

$$h_2 = \sigma(-x_1 - x_2 + 1.5) \quad \text{(NAND-Gate)} \tag{30}$$

$$\text{XOR} = \sigma(h_1 + h_2 - 1.5) \tag{31}$$

- **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

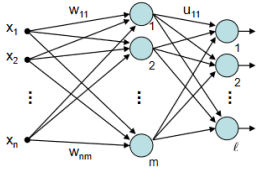
- **Multilayer Perceptron (MLP)**: Neuronales Netz mit versteckten Schichten
- Forward-Pass für 2-Schicht-Netz:

$$\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \quad \text{(lineare Transformation)} \tag{32}$$

$$\mathbf{a}^{(1)} = \sigma(\mathbf{z}^{(1)}) \quad \text{(Aktivierung)} \tag{33}$$

$$\mathbf{z}^{(2)} = \mathbf{W}^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)} \tag{34}$$

$$\hat{\mathbf{y}} = \sigma(\mathbf{z}^{(2)}) \tag{35}$$



- Mean Squared Error (MSE) Loss:

$$L(\mathbf{W}, \mathbf{b}) = \frac{1}{n}\sum_{i=1}^{n} ||\hat{\mathbf{y}}_i - \mathbf{y}_i||_2^2 \tag{36}$$

- 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

- **Verlustfunktion**:

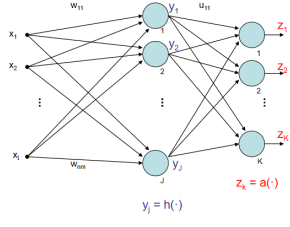
$$f(\mathbf{w}) = \sum_{\mathbf{x} \in B} \|g(\mathbf{w}; \mathbf{x}) - g^*(\mathbf{x})\|^2 \quad \rightarrow \min \tag{37}$$

- mit dem Output des Netzes $g(\mathbf{w}; \mathbf{x})$ und dem erwarteten Output $g^*(\mathbf{x})$
- **Gradientenabstieg für alle Parameter**:

$$\mathbf{u}^{(t+1)} = \mathbf{u}^{(t)} - \gamma \nabla_{\mathbf{u}} f(\mathbf{w}^{(t)}, \mathbf{u}^{(t)}) \tag{38}$$

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \gamma \nabla_{\mathbf{w}} f(\mathbf{w}^{(t)}, \mathbf{u}^{(t)}) \tag{39}$$

- $\mathbf{x}_i$: Input-Vektoren
- $\mathbf{y}_j$: Werte nach dem ersten Layer
- $\mathbf{z}_k$: Werte nach dem zweiten Layer



Backpropagation: Grundidee

- **Backpropagation** [5]:
  Effizienter Algorithmus zur Berechnung von Gradienten
- **Problem**: Wie berechnen wir $\frac{\partial L}{\partial w_{ij}}$ in tiefen Netzen?
- **Lösung**: Anwendung der **Kettenregel** der Differentiation:

$$\frac{\partial L}{\partial w_{ij}^{(\ell)}} = \frac{\partial L}{\partial z_j^{(\ell)}} \cdot \frac{\partial z_j^{(\ell)}}{\partial w_{ij}^{(\ell)}} \tag{40}$$

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

$$\delta_j^{(\ell)} = \frac{\partial L}{\partial z_j^{(\ell)}} \tag{41}$$

- **Idee**: Berechne Fehler rückwärts durch das Netz

Backpropagation: Notation und Ablauf

- **Notation**:
  - L: Verlustfunktion, $a_j^{(\ell)}$: Aktivierung von Neuron j in Schicht ℓ
  - $z_j^{(\ell)} = \sum_i w_{ij}^{(\ell)} a_i^{(\ell-1)} + b_j^{(\ell)}$: Eingabe vor Aktivierung
  - $\delta_j^{(\ell)} = \frac{\partial L}{\partial z_j^{(\ell)}}$: Lokaler Gradient (Fehlerterm)
- **Forward Pass**: Berechne Ausgaben für alle Schichten
- **Backward Pass**: Berechne Gradienten rekursiv
- **Output-Schicht**:

$$\delta_j^{(L)} = \frac{\partial L}{\partial a_j^{(L)}} \cdot \sigma'(z_j^{(L)}) \tag{42}$$

- **Versteckte Schichten**:

$$\delta_j^{(\ell)} = \left(\sum_k \delta_k^{(\ell+1)} w_{jk}^{(\ell+1)} \right) \sigma'(z_j^{(\ell)}) \tag{43}$$

Backpropagation: Gradientenberechnung

- **Gradientenberechnung**:

$$\frac{\partial L}{\partial w_{ij}^{(\ell)}} = \delta_j^{(\ell)} \cdot a_i^{(\ell-1)} \tag{44}$$

$$\frac{\partial L}{\partial b_j^{(\ell)}} = \delta_j^{(\ell)} \tag{45}$$

- **Interpretation**:
  - Gewicht-Gradient = lokaler Gradient × Eingabeaktivierung
  - Bias-Gradient = lokaler Gradient
  - Effizienz: $O(W)$ statt $O(W^2)$ für numerische Differentiation
- **Komplexität**: Linear in der Anzahl der Parameter

MLP Training: Grundlagen

- Analog zum SLP: Gradientenabstieg zur Fehlerminimierung
- **Batch-Gradient**:

$$\nabla f(w,u) = \sum_{x,z^* \in B} \nabla f(w,u;x,z^*) \tag{46}$$

- **Sigmoid-Aktivierungsfunktion**:

$$\sigma(x) = \frac{1}{1+e^{-x}} \quad \text{mit} \quad \sigma'(x) = \sigma(x) \cdot (1-\sigma(x)) \tag{47}$$

- **Kettenregel**:

$$[f(g(x))]' = f'(g(x)) \cdot g'(x) \tag{48}$$

- **Fehlerterm** (Delta): $\delta_j = \frac{\partial L}{\partial z_j}$

MLP Training: Schritt 1 - Fehlerterm Output-Schicht

- **Ziel**: Berechne wie stark jedes Output-Neuron zum Gesamtfehler beiträgt
- **Verlustfunktion**: $L = \sum_k (z_k - z_k^*)^2$ (Mean Squared Error)
- **Schritt-für-Schritt Herleitung**:
  1. **Ableitung nach Aktivierung**: $\frac{\partial L}{\partial z_k} = 2(z_k - z_k^*)$
  2. **Aktivierung**: $z_k = \sigma(u_k) = \sigma(\sum_j u_{jk} y_j)$
  3. **Sigmoid-Ableitung**: $\sigma'(u_k) = \sigma(u_k)(1 - \sigma(u_k)) = z_k(1 - z_k)$
  4. **Kettenregel**: $\frac{\partial L}{\partial u_k} = \frac{\partial L}{\partial z_k} \cdot \frac{\partial z_k}{\partial u_k}$
- **Resultat - Fehlerterm Output**:

$$\delta_k^{(out)} = \frac{\partial L}{\partial u_k} = 2(z_k - z_k^*) \cdot z_k \cdot (1 - z_k) \tag{49}$$

MLP Training: Schritt 2 - Gradienten Output-Gewichte

- **Ziel**: Berechne Gradienten für Gewichte zwischen versteckter und Output-Schicht
- **Ausgangspunkt**: Wir haben $\delta_k^{(out)} = \frac{\partial L}{\partial u_k}$
- **Schritt-für-Schritt**:
  1. **Netzinput**: $u_k = \sum_j u_{jk} \cdot y_j$ (Gewicht × Aktivierung)
  2. **Ableitung nach Gewicht**: $\frac{\partial u_k}{\partial u_{jk}} = y_j$
  3. **Kettenregel anwenden**:

$$\frac{\partial L}{\partial u_{jk}} = \frac{\partial L}{\partial u_k} \cdot \frac{\partial u_k}{\partial u_{jk}} = \delta_k^{(out)} \cdot y_j \tag{50}$$

- **Interpretation**: Gradient = Fehlerterm × Eingangssignal
- **Gewicht-Update**: $u_{jk}^{neu} = u_{jk}^{alt} - \eta \cdot \delta_k^{(out)} \cdot y_j$

MLP Training: Schritt 3 - Fehlerterm versteckte Schicht

- **Problem**: Wie berechnen wir Fehler für versteckte Neuronen?
- **Idee**: Fehler "fließt rückwärts" von Output zu versteckter Schicht
- **Schritt-für-Schritt**:
  1. **Aktivierung versteckt**: $y_j = \sigma(h_j)$ mit $h_j = \sum_i w_{ij} x_i$
  2. **Einfluss auf alle Outputs**: y_j beeinflusst alle u_k über Gewichte u_{jk}
  3. **Kettenregel für mehrere Ausgänge**:

$$\frac{\partial L}{\partial h_j} = \sum_k \frac{\partial L}{\partial u_k} \cdot \frac{\partial u_k}{\partial y_j} \cdot \frac{\partial y_j}{\partial h_j} \tag{51}$$

  4. **Einsetzen**: $\frac{\partial u_k}{\partial y_j} = u_{jk}$ und $\frac{\partial y_j}{\partial h_j} = y_j(1 - y_j)$
- **Resultat**:

$$\delta_j^{(hidden)} = y_j(1 - y_j) \sum_k \delta_k^{(out)} \cdot u_{jk} \tag{52}$$

MLP Training: Schritt 4 - Gradienten versteckte Gewichte

- **Letzter Schritt**: Gradienten für Gewichte zwischen Input und versteckter Schicht
- **Analogie zum Output**: Gleiche Logik wie bei Output-Gewichten
- **Schritt-für-Schritt**:
  1. **Wir haben**: $\delta_j^{(hidden)} = \frac{\partial L}{\partial h_j}$
  2. **Netzinput**: $h_j = \sum_i w_{ij} \cdot x_i$
  3. **Ableitung**: $\frac{\partial h_j}{\partial w_{ij}} = x_i$
  4. **Kettenregel**:

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial h_j} \cdot \frac{\partial h_j}{\partial w_{ij}} = \delta_j^{(hidden)} \cdot x_i \tag{53}$$

- **Gewicht-Update**: $w_{ij}^{neu} = w_{ij}^{alt} - \eta \cdot \delta_j^{(hidden)} \cdot x_i$
- **Backpropagation komplett**: Fehler propagiert von Output zurück zum Input!

Verallgemeinerung Training von Neuronalen Netzen: M-Layer-Perceptron

- bei einem Neuronalen Netz mit L Layern $S_1, S_2, \ldots, S_L$
- den Gewichten w_{ij} in der Matrix $\mathbf{W}$
- dem Output eines Neurons o_j
- ist der Fehlerterm:

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

- Der Korrekturterm für die einzelnen Gewichte ist dann:

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} - \gamma \cdot o_i \cdot \delta_j \tag{54}$$

Fortgeschrittene Optimierungsalgorithmen

- **Momentum**: Beschleunigung in konsistente Richtungen

$$\mathbf{v}^{(t+1)} = \beta \mathbf{v}^{(t)} + \eta \nabla L(\theta^{(t)}) \tag{55}$$

$$\theta^{(t+1)} = \theta^{(t)} - \mathbf{v}^{(t+1)} \tag{56}$$

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

$$\mathbf{m}^{(t+1)} = \beta_1 \mathbf{m}^{(t)} + (1 - \beta_1) \nabla_\theta L^{(t)} \tag{57}$$

$$\mathbf{v}^{(t+1)} = \beta_2 \mathbf{v}^{(t)} + (1 - \beta_2) (\nabla_\theta L^{(t)})^2 \tag{58}$$

$$\hat{\mathbf{m}}^{(t)} = \frac{\mathbf{m}^{(t+1)}}{1 - \beta_1^{t+1}} \quad \text{(Bias-Korrektur)} \tag{59}$$

$$\hat{\mathbf{v}}^{(t)} = \frac{\mathbf{v}^{(t+1)}}{1 - \beta_2^{t+1}} \quad \text{(Bias-Korrektur)} \tag{60}$$

$$\theta^{(t+1)} = \theta^{(t)} - \frac{\eta}{\sqrt{\hat{\mathbf{v}}^{(t)}} + \epsilon} \hat{\mathbf{m}}^{(t)} \tag{61}$$

- 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
- **Kompositionelle Struktur**: Viele reale Funktionen haben hierarchische Struktur

$$f(\mathbf{x}) = g_L(g_{L-1}(\ldots g_2(g_1(\mathbf{x}))\ldots)) \tag{62}$$

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

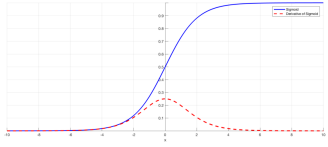
$$\mathbf{h}^{(\ell)} = \sigma(\mathbf{W}^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}) \tag{63}$$

- **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:

$$\frac{\partial L}{\partial \mathbf{W}^{(\ell)}} \propto \prod_{i=\ell+1}^{L} \mathbf{W}^{(i)} \sigma'(\mathbf{z}^{(i)}) \tag{64}$$



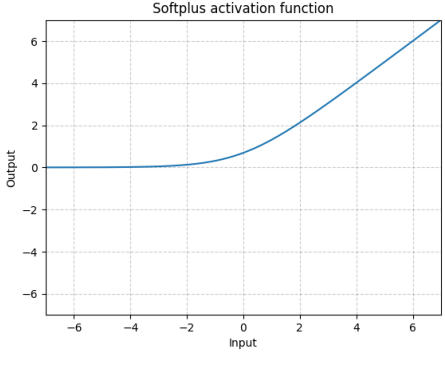
- Für $L - \ell$ Schichten: Faktor $\leq (0.25)^{L-\ell}$
- **Beispiel**: Bei 10 Schichten kann Gradient um Faktor 10^{-6} schrumpfen!
- **Lösungsansätze**:
  - ReLU-Aktivierungen: ReLU'$(x) = 1$ für $x > 0$
  - Residual Connections (ResNets)
  - Normalization (BatchNorm, LayerNorm)
  - Bessere Initialisierung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

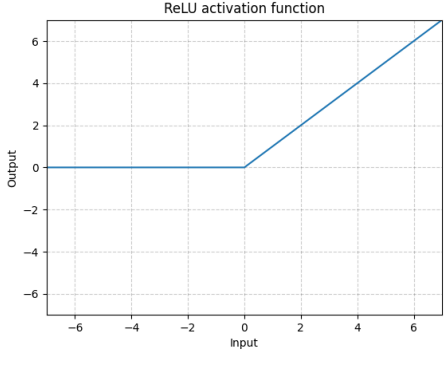
Aktivierungsfunktionen – Mathematische Eigenschaften und Warum sie wichtig sind

- **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**:

$$\text{Ohne } \sigma: f(\mathbf{x}) = \mathbf{W}_2(\mathbf{W}_1\mathbf{x}) = (\mathbf{W}_2\mathbf{W}_1)\mathbf{x} = \mathbf{W}_{\text{linear}}\mathbf{x} \tag{65}$$

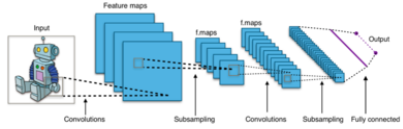
Häufige Aktivierungsfunktionen – Visualisierung

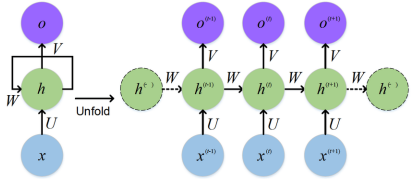




Weitere Netzwerk Architekturen

- 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)



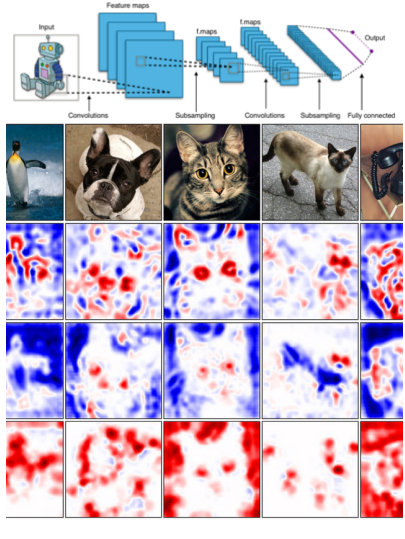


Motivation: Warum spezielle Architekturen für Bilder?

- **Problem 1: Verlust räumlicher Information**
  - Ein MLP benötigt einen flachen Input-Vektor (Flattening).
  - Ein Bild ist aber eine 2D-Matrix (Pixel-Nachbarschaften sind wichtig!).
  - Durch Flattening geht die Information ”Pixel A liegt neben Pixel B” verloren.
- **Problem 2: Parameter-Explosion (Curse of Dimensionality)**
  - Beispiel: Bild **1000 × 1000** Pixel = 1.000.000 Inputs.
  - Erste Hidden Layer mit 1.000 Neuronen.
  - Anzahl Gewichte: $10^6 \times 10^3 = 10^9$ (1 Milliarde Parameter!)
  - **Folge**: Unmöglich zu trainieren, extremes Overfitting.
- **Lösung der Natur (Visueller Kortex)**:
  - **Lokale Rezeptive Felder**: Neuronen schauen nur auf kleine Ausschnitte.
  - **Translation Invariance**: Ein ”Kanten-Detektor” funktioniert oben links genauso wie unten rechts.

Was sind Convolutional Neural Networks (CNNs)?

- **CNNs**: Speziell für räumliche Daten entwickelte neuronale Netze
- **Hauptanwendung**: Bildverarbeitung, Computer Vision
- **Grundidee**: Nutze lokale Strukturen in Bildern
- **Kernoperationen**:
  - **Convolution**: Filtere lokale Features
  - **Pooling**: Reduziere räumliche Dimensionen
  - **Fully Connected**: Klassifikation am Ende
- **Hierarchische Feature-Extraktion**:
  - **Layer 1**: Kanten, Ecken
  - **Layer 2**: Texturen, Formen
  - **Layer 3**: Objektteile
  - **Layer 4**: Vollständige Objekte
- **Vorteil**: Automatisches Lernen relevanter Features



CNN Grundoperationen: Convolution

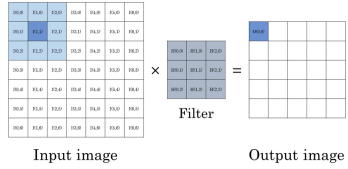
- **Convolution Operation**:

$$(f * g)(x, y) = \sum_m \sum_n f(m, n) \cdot g(x - m, y - n) \tag{66}$$

- **In CNNs**:

$$\text{Output}(i, j) = \sum_m \sum_n \text{Filter}(m, n) \cdot \text{Input}(i + m, j + n) \tag{67}$$

- **Filter (Kernel)**:
  - Kleine Matrizen (z.B. 3×3, 5×5)
  - Erkennen spezifische Muster
  - Gewichte werden gelernt
- **Beispiel**: Kantendetektor, Blur-Filter
- **Feature Maps**: Ausgabe nach Convolution + Aktivierung



CNN Grundoperationen: Pooling

- **Was ist Pooling?** Reduziert die räumliche Größe der Feature Maps
- **Ziele**:
  - **Dimensionsreduktion**: Weniger Parameter, schnellere Berechnung
  - **Translation Invarianz**: Kleine Verschiebungen werden ignoriert
  - **Überanpassung reduzieren**: Weniger Details = bessere Generalisierung
- **Max Pooling** (häufigste Variante):

$$\text{MaxPool}(\mathbf{X})_{i,j} = \max_{(p,q)\in N_{i,j}} \mathbf{X}_{p,q} \tag{68}$$

- **Average Pooling**:

$$\text{AvgPool}(\mathbf{X})_{i,j} = \frac{1}{|N_{i,j}|} \sum_{(p,q)\in N_{i,j}} \mathbf{X}_{p,q} \tag{69}$$

- **Typische Parameter**: 2×2 Fenster mit Stride 2 → Halbiert Dimensionen
- **Keine lernbaren Parameter**: Pooling-Operation ist fest definiert

- **Beispiel: 2×2 Max Pooling**
- **Input (4×4)**:

$$\begin{pmatrix} 1 & 3 & 2 & 4 \\ 5 & 6 & 1 & 2 \\ 3 & 2 & 4 & 7 \\ 1 & 0 & 3 & 5 \end{pmatrix}$$

- **Output (2×2)**:

$$\begin{pmatrix} 6 & 4 \\ 3 & 7 \end{pmatrix}$$

- **Jeder Wert** = Maximum des 2×2 Bereichs

Warum funktionieren CNNs? – Mathematische Prinzipien

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

$$y_{i,j}^{(\ell)} = \sigma\left(\sum_{m=-k}^{k}\sum_{n=-k}^{k} w_{m,n}^{(\ell)} \cdot x_{i+m,j+n}^{(\ell-1)} + b^{(\ell)}\right) \tag{70}$$

- **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 $\mathbf{t}$

$$\text{Conv}(T_{\mathbf{t}}[\mathbf{x}]) = T_{\mathbf{t}}[\text{Conv}(\mathbf{x})] \tag{71}$$

- **Pooling** führt zu begrenzter Translationsinvarianz:

$$\text{MaxPool}(\mathbf{X})_{i,j} = \max_{(p,q)\in N_{i,j}} \mathbf{X}_{p,q} \tag{72}$$

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

Spezielle Netzwerkarchitekturen: CNN – Implementierung

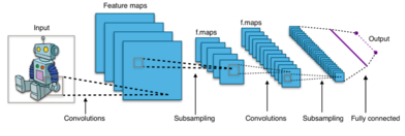
- **Convolutional Neural Networks** [8]: Spezialisiert auf gitterförmige Daten (Bilder)
- **Faltungsoperation** (2D-Convolution):

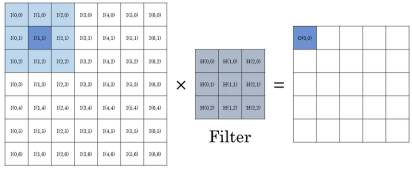
$$(\mathbf{I} * \mathbf{K})_{i,j} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \mathbf{I}_{i+m,j+n} \cdot \mathbf{K}_{m,n} \tag{73}$$

- $\mathbf{I}$: Input-Feature-Map, $\mathbf{K}$: Kernel (Filter) der Größe $M \times N$
- **Pooling**: Dimensionsreduktion, z.B. Max-Pooling:

$$\text{MaxPool}(\mathbf{X})_{i,j} = \max_{p,q \in P_{i,j}} \mathbf{X}_{p,q} \tag{74}$$

- **Parameter Sharing**: Derselbe Filter wird über gesamte Feature-Map angewendet
- **Translation Invariance**: Robustheit gegenüber Verschiebungen





CNN Training: Schritt 1a - Notation und Setup

- **Ziel**: Berechne Ausgabe des CNN für einen Input
- **Input**:
  - Bild $\mathbf{X}^{(0)} \in \mathbb{R}^{H \times W \times C}$
  - Dimensionen: Höhe × Breite × Kanäle
- **Notation**:
  - $\mathbf{W}^{(\ell)}_{m,n,c,k}$: Filter-Gewicht an Position (m, n)
  - Indizes: Input-Kanal c, Output-Kanal k
  - K: Filter-Größe (z.B. 3×3)
  - $C^{(\ell)}$: Anzahl Kanäle in Schicht ℓ

CNN Training: Schritt 1b - Forward Pass Berechnung

Schritt-für-Schritt Berechnung:

1. **Convolution Layer ℓ**:

$$\mathbf{Z}^{(\ell)}_{i,j,k} = \sum_{c=1}^{C^{(\ell-1)}} \sum_{m=1}^{K} \sum_{n=1}^{K} \mathbf{W}^{(\ell)}_{m,n,c,k} \cdot \mathbf{X}^{(\ell-1)}_{i+m-1,j+n-1,c} + b^{(\ell)}_{k}$$

2. **Aktivierung**:
   $\mathbf{X}^{(\ell)}_{i,j,k} = \text{ReLU}(\mathbf{Z}^{(\ell)}_{i,j,k}) = \max(0, \mathbf{Z}^{(\ell)}_{i,j,k})$
3. **Pooling**:
   $\mathbf{P}^{(\ell)}_{i,j,k} = \max_{(p,q)\in N_{i,j}} \mathbf{X}^{(\ell)}_{p,q,k}$
4. **Fully Connected**:
   Flatten → $\mathbf{y} = \text{Softmax}(\mathbf{W}_{fc}\mathbf{h} + \mathbf{b}_{fc})$

CNN Training: Schritt 2a - Verlust und Output-Layer

- **Verlustfunktion**:
  - Wir verwenden meist Cross-Entropy für Klassifikation.
  - Sei N die Batch-Größe und C die Anzahl der Klassen:

$$L = -\sum_{i=1}^{N}\sum_{c=1}^{C} y_{i,c}\log(\hat{y}_{i,c}) \tag{75}$$

- **Gradient für Output-Layer (Fully Connected)**:
  - Dies ist identisch zum Standard-MLP (Backpropagation).
  - Der Fehlerterm δ am Ausgang ist die Differenz zwischen Vorhersage und Label:

$$\frac{\partial L}{\partial \mathbf{W}_{fc}} = (\hat{\mathbf{y}} - \mathbf{y}_{true})\mathbf{h}^T \tag{76}$$

CNN Training: Schritt 2b - Gradienten für Filter

- **Herausforderung**:
  - Wir haben Shared Weights! Ein Filter wird an vielen Positionen angewendet.
  - Der Gradient für ein Filtergewicht ist die Summe der Gradienten von allen Positionen, wo es angewendet wurde.
- **Gradient für Convolution-Filter**:

$$\frac{\partial L}{\partial \mathbf{W}^{(\ell)}_{m,n,c,k}} = \sum_{i,j} \underbrace{\frac{\partial L}{\partial \mathbf{Z}^{(\ell)}_{i,j,k}}}_{\text{Fehler an Output-Position}} \cdot \underbrace{\mathbf{X}^{(\ell-1)}_{i+m-1,j+n-1,c}}_{\text{Input an entsprechender Position}} \tag{77}$$

- **Intuition**: Korrelation zwischen dem incoming Fehler und dem lokalen Input-Patch.

CNN Training: Schritt 3 - Backpropagation durch Convolution

- **Problem**: Wie berechnen wir $\frac{\partial L}{\partial \mathbf{X}^{(\ell-1)}}$ aus $\frac{\partial L}{\partial \mathbf{Z}^{(\ell)}}$?
- **Convolution ist linear**: Können Kettenregel anwenden
- **Schritt-für-Schritt Herleitung**:
  1. **Forward Pass** (zur Erinnerung):
     $\mathbf{Z}^{(\ell)}_{i,j,k} = \sum_{c,m,n} \mathbf{W}^{(\ell)}_{m,n,c,k} \cdot \mathbf{X}^{(\ell-1)}_{i+m-1,j+n-1,c}$
  2. **Ableitung nach Input**:
     $$\frac{\partial \mathbf{Z}^{(\ell)}_{i,j,k}}{\partial \mathbf{X}^{(\ell-1)}_{p,q,c}} = \mathbf{W}^{(\ell)}_{p-i+1,q-j+1,c,k}$$
  3. **Kettenregel**:
     $$\frac{\partial L}{\partial \mathbf{X}^{(\ell-1)}_{p,q,c}} = \sum_{i,j,k} \frac{\partial L}{\partial \mathbf{Z}^{(\ell)}_{i,j,k}} \cdot \mathbf{W}^{(\ell)}_{p-i+1,q-j+1,c,k}$$
- **Interpretation**: Backpropagation = Convolution mit "umgedrehten" Filtern
- **Implementierung**: Oft als "Transposed Convolution" bezeichnet

CNN Training: Schritt 4 - Backpropagation durch Pooling

- **Max Pooling Backpropagation**: Nur das Maximum bekommt den Gradienten
- **Schritt-für-Schritt für Max Pooling**:
  1. **Forward**: $\mathbf{P}_{i,j,k} = \max_{(p,q)\in N_{i,j}} \mathbf{X}_{p,q,k}$
  2. **Speichere Positionen**: Merke dir (p^*, q^*) wo Maximum war
  3. **Backward**:

$$\frac{\partial L}{\partial \mathbf{X}_{p,q,k}} = \begin{cases} \frac{\partial L}{\partial \mathbf{P}_{i,j,k}} & \text{wenn } (p,q) = (p^*, q^*) \\ 0 & \text{sonst} \end{cases} \tag{78}$$

- **Average Pooling**: Gradient wird gleichmäßig verteilt

$$\frac{\partial L}{\partial \mathbf{X}_{p,q,k}} = \frac{1}{|N_{i,j}|} \frac{\partial L}{\partial \mathbf{P}_{i,j,k}} \tag{79}$$

- **Praktische Implementierung**: "Switch Variables" speichern Max-Positionen
- **Resultat**: Gradienten fließen nur zu den "wichtigsten" Neuronen zurück

CNN Training: Schritt 5 - Optimierung und praktische Aspekte

- **Gradient Descent Update**: Wie bei MLPs, aber für Filter-Gewichte

$$\mathbf{W}_{new}^{(\ell)} = \mathbf{W}_{old}^{(\ell)} - \eta \frac{\partial L}{\partial \mathbf{W}^{(\ell)}} \tag{80}$$

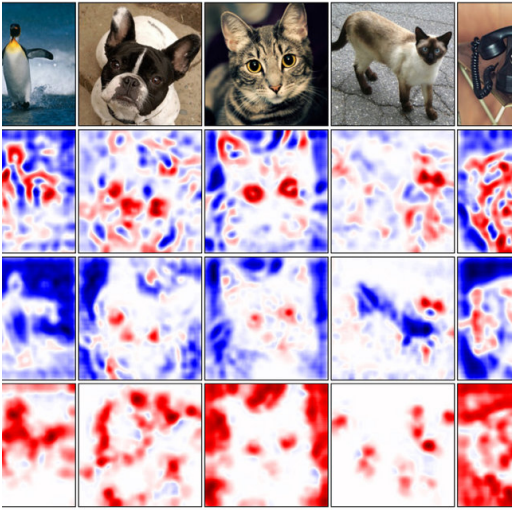
- **Batch Processing**: Trainiere mit Mini-Batches für Effizienz

$$\frac{\partial L_{batch}}{\partial \mathbf{W}^{(\ell)}} = \frac{1}{B} \sum_{b=1}^{B} \frac{\partial L_b}{\partial \mathbf{W}^{(\ell)}} \tag{81}$$

- **Besondere Herausforderungen bei CNNs**:
  - **Speicher**: Feature Maps können sehr groß werden
  - **Initialisierung**: Filter richtig initialisieren
  - **Learning Rate**: Oft kleiner als bei MLPs
  - **Data Augmentation**: Rotation, Skalierung, Crop für mehr Daten
- **Moderne Optimierer**: Adam, RMSprop statt SGD
- **Regularisierung**: Dropout, Batch Normalization, Weight Decay

Spezielle Netzwerkarchitekturen: CNN

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



Motivation: Die Evolution von CNN-Architekturen

- **Warum schauen wir uns ”alte” Netzwerke an?**
  - Moderne Architekturen (wie ConvNeXt, EfficientNet) basieren auf Ideen, die über Jahre entwickelt wurden.
  - Um zu verstehen, warum Netze heute so aussehen, muss man die Evolutionsschritte kennen.
- **Die drei Äras des CNN-Designs**:
  - **Die Pionier-Phase (1998-2012)**: LeNet, AlexNet. Beweis, dass es überhaupt funktioniert. Fokus auf Tiefe und GPUs.
  - **Die Vertiefungs-Phase (2014-2015)**: VGG, Inception, ResNet. ”Deeper is better”, aber wie trainieren wir das? (Skip Connections, kleine Filter).
  - **Die Effizienz-Phase (2017-heute)**: MobileNet, EfficientNet, Transformers. ”Smarter is better”. Fokus auf Rechenleistung und Parameter-Effizienz.

Historische CNN-Architekturen: AlexNet (2012) [10]

- **Durchbruch**: Gewinner ImageNet Challenge 2012 (Top-5 Error: 15.3%)
- **Architektur-Details**:
  - 8 Schichten (5 Conv + 3 FC), 60 Millionen Parameter
  - Input: 224×224×3, Output: 1000 Klassen
  - Erste GPU-Implementation für Deep Learning
- **Schicht-Struktur**:

$$\text{Input} \xrightarrow{\text{Conv1: } 11\times11,\ 96} \text{ReLU} \xrightarrow{\text{MaxPool: } 3\times3} \text{Conv2: } 5\times5,\ 256 \rightarrow \cdots \rightarrow \text{FC: } 4096 \tag{82}$$

- **Innovationen**:
  - **ReLU-Aktivierung**: Statt Sigmoid/Tanh → schnelleres Training
  - **Dropout** [9]: Regularisierung in FC-Layern (p=0.5)
  - **Data Augmentation**: Random Crops, Horizontal Flips
  - **Local Response Normalization**: Normalisierung zwischen Kanälen
- **Bedeutung**: Bewies Potenzial von Deep CNNs für Computer Vision

Moderne CNN-Architekturen: VGGNet (2014) [11]

- **Philosophie**: "Deeper is better" - sehr tiefe Netzwerke (16-19 Schichten)
- **Architektur-Prinzip**: Nur 3×3 Convolutions, 2×2 Max Pooling
- **VGG-16 Struktur**:

$$\begin{aligned}
&\text{Block 1: } 2 \times (\text{Conv } 3\times3,\ 64) \rightarrow \text{MaxPool}\\
&\text{Block 2: } 2 \times (\text{Conv } 3\times3,\ 128) \rightarrow \text{MaxPool}\\
&\text{Block 3: } 3 \times (\text{Conv } 3\times3,\ 256) \rightarrow \text{MaxPool}\\
&\text{Block 4: } 3 \times (\text{Conv } 3\times3,\ 512) \rightarrow \text{MaxPool}\\
&\text{Block 5: } 3 \times (\text{Conv } 3\times3,\ 512) \rightarrow \text{MaxPool}\\
&\text{FC: } 4096 \rightarrow 4096 \rightarrow 1000
\end{aligned} \tag{83}$$

- **Warum 3×3 Filter?**
  - Zwei 3×3 Filter = ein 5×5 Filter (gleiche rezeptive Feldgröße)
  - Weniger Parameter: $2 \times (3^2 \times C^2) < 5^2 \times C^2$
  - Mehr Nichtlinearitäten durch zusätzliche ReLU-Schichten
- **Problem**: Sehr viele Parameter (138M), langsames Training

Revolutionäre CNN-Architekturen: ResNet (2015) [13]

- **Problem**: Verschwindende Gradienten bei sehr tiefen Netzwerken (>20 Schichten)
- **Lösung**: Residual Connections (Skip Connections)
- (Schmidhuber notes: Highway Networks [12] enabled very deep networks using similar principles.)
- **Residual Block**:

$$\mathbf{y} = F(\mathbf{x}, \{W_i\}) + \mathbf{x} \tag{84}$$

$$\mathbf{H}(\mathbf{x}) = F(\mathbf{x}) + \mathbf{x}, \text{ wo } F(\mathbf{x}) = W_2\sigma(W_1\mathbf{x}) \tag{85}$$

- **Intuition**: Lerne die Residual-Funktion $F(\mathbf{x}) = \mathbf{H}(\mathbf{x}) - \mathbf{x}$
- **Gradient-Flow**: Skip Connections ermöglichen direkten Gradient-Fluss

$$\frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{y}} \left(\frac{\partial F}{\partial \mathbf{x}} + \mathbf{I} \right) \tag{86}$$

- **Varianten**: ResNet-34, ResNet-50, ResNet-101, ResNet-152
- **Bottleneck Design** (ResNet-50+): 1×1 → 3×3 → 1×1 für Effizienz
- **Erfolg**: Ermöglichte Training von 152-Schicht-Netzwerken

Effiziente CNN-Architekturen: MobileNet & EfficientNet

- **MobileNet (2017)** [14]: Optimiert für mobile Geräte
  - **Depthwise Separable Convolution**:

$$\text{Standard Conv: } M \times N \times D_K \times D_K \text{ Parameter} \tag{87}$$

$$\text{Depthwise Sep: } M \times D_K \times D_K + M \times N \text{ Parameter} \tag{88}$$

  - **Reduktion**: Faktor $\frac{1}{N} + \frac{1}{D_K^2}$ (z.B. 8-9× weniger Parameter)
  - **Width Multiplier**: $\alpha \in (0, 1]$ für Modell-Skalierung
- **EfficientNet (2019)** [15]: Systematisches Netzwerk-Scaling
  - **Compound Scaling**: Gleichzeitige Skalierung von Tiefe, Breite, Auflösung

$$\text{depth: } d = \alpha^\phi, \quad \text{width: } w = \beta^\phi, \quad \text{resolution: } r = \gamma^\phi \tag{89}$$

  - **Constraint**: $\alpha \cdot \beta^2 \cdot \gamma^2 \approx 2$, $\alpha \geq 1, \beta \geq 1, \gamma \geq 1$
  - **EfficientNet-B0 bis B7**: Verschiedene Skalierungsstufen

Vision Transformer: CNN-Alternative (2020) [16]

- **Paradigmenwechsel**: Transformer-Architektur für Computer Vision
- **Grundidee**: Bild als Sequenz von Patches behandeln
- **Patch Embedding**:

$$\text{Image } \mathbf{X} \in \mathbb{R}^{H \times W \times C} \rightarrow \text{Patches } \mathbf{x}_p \in \mathbb{R}^{N \times (P^2 \cdot C)} \tag{90}$$

$$\mathbf{z}_0 = [\mathbf{x}_{class}; \mathbf{x}_p^1\mathbf{E}; \mathbf{x}_p^2\mathbf{E}; \ldots; \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{pos} \tag{91}$$

- **Self-Attention**: Patches ”kommunizieren” miteinander

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{92}$$

- **Vorteile**:
  - Globale Aufmerksamkeit von Anfang an (vs. lokale CNN-Filter)
  - Bessere Skalierung mit Datenmengen
  - Transfer Learning von NLP
- **Nachteile**: Benötigt sehr große Datenmengen, weniger induktive Biases

CNN-Architekturen: Zusammenfassung und Trends

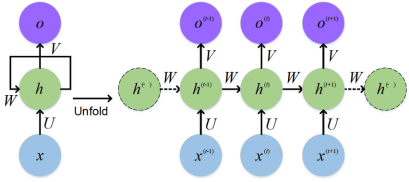
- **Entwicklung der CNN-Architekturen**:
  - **2012 AlexNet**: Beweis dass Deep Learning funktioniert
  - **2014 VGGNet**: Deeper Networks, kleinere Filter
  - **2015 ResNet**: Skip Connections lösen Vanishing Gradient Problem
  - **2017 MobileNet**: Effizienz für mobile Anwendungen
  - **2019 EfficientNet**: Systematisches Model Scaling
  - **2020 Vision Transformer**: Alternative zu CNNs
- **Aktuelle Trends**:
  - **Hybrid-Architekturen**: CNNs + Transformers (ConvNeXt, CoAtNet)
  - **Neural Architecture Search (NAS)**: Automatisches Design
  - **Self-Supervised Learning**: Weniger gelabelte Daten
  - **Multimodal Models**: Vision + Language (CLIP, DALL-E)
- **Auswahlkriterien**:
  - **Accuracy**: ResNet, EfficientNet, Vision Transformer
  - **Speed**: MobileNet, ShuffleNet
  - **Memory**: MobileNet, SqueezeNet
  - **Interpretability**: Klassische CNNs mit Attention

Motivation: Warum Sequenzmodelle?

- **Das Limit von Feedforward Netzen (MLPs)**:
  - Erwarten Eingabe fester Größe (z.B. genau 100 Features).
  - Haben kein ”Gedächtnis” für Vorheriges (stateless).
  - Betrachten alle Eingaben gleichzeitig, nicht nacheinander.
- **Realität sequentieller Daten**:
  - Text, Audio, Zeitreihen, Video haben zeitliche Abhängigkeiten.
  - Kontext ist entscheidend: ”Die Bank am Fluss” vs. ”Die Bank zahlt Zinsen”.
  - Variable Länge der Eingabe.
- **Idee der Rekurrenz**:
  - Das Netzwerk verarbeitet Input Schritt für Schritt.
  - Es erhält einen ”internen Zustand” (Memory), der Kontext speichert.
  - **Recycling**: Wir nutzen **dieselben Gewichte** für jeden Zeitschritt!

Spezielle Netzwerkarchitekturen: RNN

- Anschaulich: Schleife in Netzwerk ”merkt” sich vorherige Zustände (Elman Networks [17])
- funktioniert für kurze Zeiträume
- Entfaltung eines RNN →viele zu trainierende Gewichte
- Problem: langfristige Zusammenhänge werden nicht erfasst
- **Problem: Verschwindende Gradienten** [18, 19]:
  - Backpropagation Through Time (BPTT) multipliziert Gewichtsmatrix W oft hintereinander (W^t).
  - Wenn |Eigenwert| < 1 → Gradient geht exponentiell gegen 0.
  - Wenn |Eigenwert| > 1 → Explodierende Gradienten.
  - Frühe Schichten lernen nichts mehr.

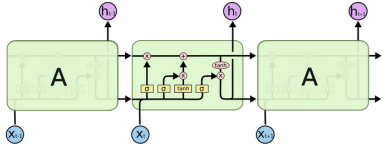


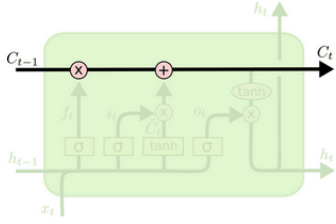
Training von RNNs: Backpropagation Through Time (BPTT)

- **Konzept** [20] (vgl. [21]): Ein RNN über T Zeitschritte ist äquivalent zu einem Feedforward-Netzwerk mit T Schichten (unrolled).
- **Shared Weights**: Die Gewichtsmatrix W ist in jedem "Zeitschritt-Layer" identisch.
- **Ablauf**:
  1. **Forward Pass**: Berechne Zustände h_t und Outputs y_t sequentiell von $t = 0$ bis T.
  2. **Loss**: Berechne Gesamtfehler (z.B. Summe der Fehler zu jedem Zeitpunkt).
  3. **Backward Pass**: Propagiere den Fehler vom letzten Zeitschritt T rückwärts durch die Zeit bis $t = 0$.
- Dies erklärt, warum Gradientenprobleme so gravierend sind: Der Fehler muss durch sehr viele Multiplikationen mit W zurücklaufen.

Spezielle Netzwerkarchitekturen: LSTM

- **Long Short-Term Memory** [22]: 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?

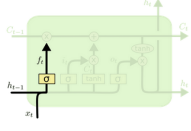




LSTM: Mathematische Formulierung

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

$$\mathbf{f}_t = \sigma(\mathbf{W}_f \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f) \tag{93}$$

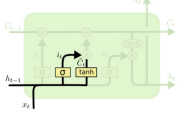


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

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

$$\mathbf{i}_t = \sigma(\mathbf{W}_i \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i) \tag{94}$$

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{W}_C \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_C) \tag{95}$$



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

- **Cell State Update**:

$$\mathbf{C}_t = \mathbf{f}_t \odot \mathbf{C}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{C}}_t \tag{96}$$

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

$$\mathbf{o}_t = \sigma(\mathbf{W}_o \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o) \tag{97}$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{C}_t) \tag{98}$$

LSTM: Cell State Update und Output

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

$$\mathbf{C}_t = \mathbf{f}_t \odot \mathbf{C}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{C}}_t \tag{99}$$

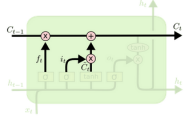
- **Output Gate**: Bestimmt, welche Teile des Cell States ausgegeben werden

$$\mathbf{o}_t = \sigma(\mathbf{W}_o \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o) \tag{100}$$

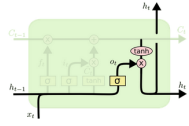
- **Hidden State**: Gefilterte Version des Cell States

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{C}_t) \tag{101}$$

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



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



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

Ausblick: Grenzen von RNNs und die Attention Revolution

- **Das Flaschenhals-Problem von LSTMs**:
  - Auch LSTMs müssen den gesamten Kontext in einen Vektor fester Größe quetschen.
  - Bei sehr langen Sequenzen (z.B. Dokumenten) gehen Details verloren.
- **Der Paradigmenwechsel: Attention Mechanism**:
  - **Idee**: Statt sich alles zu merken, lernt das Netz, bei jedem Schritt auf die relevanten Teile des Inputs zurückzuschauen.
  - Analogie: Beim Übersetzen eines Satzes schaut man für das Wort "Bank" gezielt auf den Kontext "Fluss", um es korrekt zu übersetzen.
- **Transformers**:
  - "Attention is All You Need" (2017) [23]
  - (Schmidhuber notes: "Fast Weight Programmers" (1992) [24] pioneered the linear attention mechanism.)
  - Basis für alle modernden LLMs (GPT, Llama, Claude).
- **Hinweis**: Detaillierte Behandlung von Attention und Transformers erfolgt im Modul **"Generative AI / LLMs"** im nächsten Semester!

Spezielle Netzwerkarchitekturen: Autoencoder - Motivation

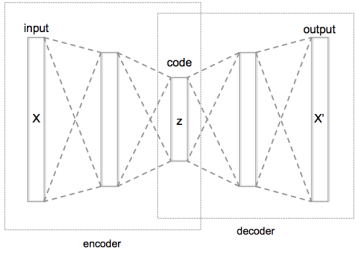
- **Das Problem mit Labels**:
  - Supervised Learning braucht gelabelte Daten (x, y).
  - Labels sind teuer und rar. Unbekannte Daten sind massenhaft vorhanden.
- **Idee des Unsupervised Learning**:
  - Können wir das Netzwerk zwingen, die **Struktur** der Daten zu lernen?
  - Aufgabe: "Lerne, den Input x auf sich selbst abzubilden: $f(x) \approx x$".
- **Der Trick mit dem Flaschenhals (Bottleneck)**:
  - Wenn wir einfach $f(x) = x$ (Identität) lernen, passiert nichts Spannendes.
  - Wir zwingen die Daten durch eine **sehr kleine Schicht** in der Mitte.
  - Analogie: Ein Buch in einen Absatz zusammenfassen (Encoder) und aus dem Absatz das Buch rekonstruieren (Decoder).
  - Nur die allerwichtigsten Informationen (Latent Features) überleben den Flaschenhals.

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

$$L(\mathbf{x}, \hat{\mathbf{x}}) = ||\mathbf{x} - \hat{\mathbf{x}}||^2 \tag{102}$$

- **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



Exkurs: Statistische Grundlagen & Bayesian Deep Learning

- **Bayesian Neural Networks (BNN)** [25]:
  - Gewichte sind keine festen Zahlen, sondern Wahrscheinlichkeitsverteilungen.
  - **Interpretation**: Ein BNN verhält sich wie ein unendliches Ensemble möglicher Modelle.
  - **Vorteil**: Durch mehrere Forward-Passes (Sampling) erhält man eine **Unsicherheitsschätzung** der Vorhersage.

- **Wichtige Begriffe für Variational Autoencoders (VAEs)**:
  - **Prior** $p(z)$: A-priori-Wissen über die latenten Variablen (z. B. $z \sim N(0, I)$). Was wir annehmen, bevor wir Daten sehen.
  - **Likelihood** $p(x|z)$: Wie wahrscheinlich sind die Daten x, gegeben die latente Variable z? (Decoder-Perspektive).
  - **Posterior** $p(z|x)$: Wahrscheinlichkeitsverteilung von z, nachdem wir das Datum x beobachtet haben. (Encoder-Ziel).

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 nicht berechenbar
- **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)**:

$$\log p(\mathbf{x}) \geq \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})] - D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p(\mathbf{z})) \tag{103}$$

- **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**: $q_\phi(\mathbf{z}|\mathbf{x}) = N(\mu_\phi(\mathbf{x}), \text{diag}(\sigma^2_\phi(\mathbf{x})))$
- **Decoder**: $p_\theta(\mathbf{x}|\mathbf{z}) = N(\mu_\theta(\mathbf{z}), \sigma^2_\theta \mathbf{I})$
- **KL-Divergenz** (analytisch berechenbar für Gauß'sche Verteilungen):

$$D_{KL} = \frac{1}{2}\sum_{j=1}^{d}(1 + \log(\sigma_j^2) - \mu_j^2 - \sigma_j^2) \tag{104}$$

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**:

$$\epsilon \sim N(\mathbf{0}, \mathbf{I}) \quad \text{(deterministisches Rauschen)} \tag{105}$$

$$\mathbf{z} = \mu + \sigma \odot \epsilon \quad \text{(deterministische Transformation)} \tag{106}$$

- **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

$$\mathbf{z}_{interp} = \alpha \mathbf{z}_1 + (1 - \alpha)\mathbf{z}_2, \quad \alpha \in [0, 1] \tag{107}$$

- **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

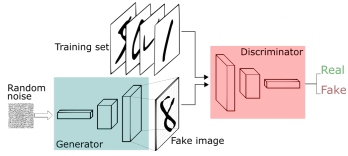
Generative Adversarial Networks (GANs): Motivation

- **Die Idee vom perfekten Fälscher**:
  - VAEs erzeugen oft verschwommene Bilder (wegen Mean Squared Error).
  - Wie erzeugen wir **gestochen scharfe** Bilder?
  - **Idee**: Anstatt eine statische Verlustfunktion zu nutzen, **lernen** wir die Verlustfunktion!
- **Das Konzept des "Adversarial Training"**:
  - Wir stellen zwei Netzwerke gegeneinander auf (wie in einem Spiel).
  - (Schmidhuber notes: The adversarial principle was introduced as "Artificial Curiosity" [26].)
  - Ein Netz versucht zu fälschen, das andere versucht, die Fälschung zu entlarven.
  - Durch diesen Wettbewerb schaukeln sich beide hoch.
- **Anwendungen**:
  - Fotorealistische Gesichter (siehe `thispersondoesnotexist.com`).
  - Super-Resolution (Upscaling von alten Filmen/Fotos).
  - Style Transfer (Mache aus einem Foto ein Gemälde).

Generative Adversarial Networks (GANs): Grundprinzip [27]

- **Grundidee**: Zwei neuronale Netze konkurrieren miteinander
  - **Generator** G: Erzeugt "gefälschte" Daten aus Rauschen
  - **Discriminator** D: Unterscheidet echte von gefälschten Daten
- **Adversarial Training**: Spieltheoretischer Ansatz

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))] \quad (108)$$



- **Training Process**:
  1. **Discriminator Training**: Maximiere $V(D,G)$
     - Lerne echte Daten als "echt" zu klassifizieren
     - Lerne generierte Daten als "gefälscht" zu erkennen
  2. **Generator Training**: Minimiere $V(D,G)$
     - Erzeuge Daten, die den Discriminator "täuschen"

GAN Training: Mathematische Details

- **Discriminator Loss** (Binary Cross-Entropy):

$$L_D = -\mathbb{E}_{x \sim p_{data}}[\log D(x)] - \mathbb{E}_{z \sim p_z}[\log(1 - D(G(z)))] \tag{109}$$

- **Generator Loss** (Ursprünglich):

$$L_G = \mathbb{E}_{z \sim p_z}[\log(1 - D(G(z)))] \tag{110}$$

- **Problem**: Vanishing Gradients bei schlechtem Generator
- **Praktische Generator Loss** (Non-saturating):

$$L_G = -\mathbb{E}_{z \sim p_z}[\log D(G(z))] \tag{111}$$

- **Nash-Gleichgewicht**: Theoretisch optimal wenn:

$$D^*(x) = \frac{p_{data}(x)}{p_{data}(x) + p_g(x)} = \frac{1}{2} \tag{112}$$

wenn $p_g = p_{data}$ (Generator erzeugt perfekte Datenverteilung)

GAN Varianten und Verbesserungen

- **Deep Convolutional GANs (DCGANs)** [28]:
  - Verwendung von Convolutional Layers
  - Batch Normalization, LeakyReLU
  - Bessere Stabilität für Bildgenerierung
- **Wasserstein GAN (WGAN)** [29]:
  - Earth Mover Distance statt JS-Divergenz
  - Stabileres Training, bessere Konvergenz-Eigenschaften
  - **Lipschitz-Constraint**: Kritisch für WGAN-Theorie
- **Conditional GANs (cGANs)** [30]:
  - Bedingung auf Labels: $G(z|y)$, $D(x|y)$
  - Kontrollierte Generierung spezifischer Klassen
- **StyleGAN** [31]:
  - Latent Space Manipulation
  - Progressive Growing, Style Transfer
  - State-of-the-art für hochauflösende Gesichter

WGAN und die Lipschitz-Constraint: Kurz erklärt

- **Problem bei Standard GANs**: Training instabil, Gradients verschwinden
- **WGAN Lösung**: Verwende Wasserstein-Distanz statt JS-Divergenz
- **Lipschitz-Constraint**:
  - **Einfach gesagt**: Discriminator darf nicht "zu steil" werden
  - **Mathematisch**: $|f(x_1) - f(x_2)| \leq L \cdot |x_1 - x_2|$
  - **Warum nötig?** Wasserstein-Distanz erfordert beschränkte Funktionen
- **Praktische Umsetzung**:
  - **Weight Clipping**: Einfach, aber problematisch
  - **Gradient Penalty**: Moderne Lösung - bestraft zu steile Gradienten
  - **Spectral Normalization**: Normalisiert Netzwerk-Gewichte
- **Resultat**: Stabileres Training, bessere Konvergenz
- **Take-away**: Theoretische Constraints führen zu praktischen Verbesserungen

GAN Herausforderungen und Lösungsansätze

- **Training Instabilität**:
  - **Problem**: Discriminator wird zu gut → Generator bekommt keine Gradienten
  - **Lösung**: Balanced Training, Learning Rate Scheduling
- **Mode Collapse**:
  - **Problem**: Generator erzeugt nur wenige Modi der Datenverteilung
  - **Lösung**: Unrolled GANs, Diversity-encouraging Loss Terms
- **Evaluation Metrics**:
  - **Inception Score (IS)**: Misst Bildqualität und Diversität
  - **Fréchet Inception Distance (FID)**: Vergleicht Feature-Statistiken
  - **Precision & Recall**: Qualität vs. Diversität Trade-off
- **Praktische Tipps**:
  - Feature Matching, Experience Replay
  - Spectral Normalization, Self-Attention
  - Progressive Growing für hohe Auflösungen

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

Wann Deep Learning? Wann klassisches Machine Learning?

- **Deep Learning ist sinnvoll bei**:
  - **Große Datenmengen**: > 10.000 Samples (je mehr, desto besser)
  - **Komplexe Muster**: Bilder, Audio, Text, Sequenzen
  - **Hierarchische Strukturen**: Features müssen automatisch gelernt werden
  - **Raw Data**: Wenig/keine Feature-Engineering nötig
  - **End-to-End Learning**: Von Rohdaten zur Entscheidung
  - **Ausreichend Rechenkapazität**: GPUs verfügbar
- **Klassisches ML ist besser bei**:
  - **Kleine Datenmengen**: < 1.000 Samples
  - **Strukturierte/tabellarische Daten**: Features sind bereits bekannt
  - **Interpretierbarkeit wichtig**: Entscheidungen müssen erklärbar sein
  - **Schnelle Inferenz**: Real-time Anwendungen mit Latenz-Constraints
  - **Begrenzte Ressourcen**: Wenig Rechenleistung/Speicher
  - **Gut definierte Features**: Domain-Wissen kann genutzt werden

Praktische Entscheidungshilfe: Deep Learning vs. klassisches ML

- **Beispiele für Deep Learning**:
  - **Computer Vision**: Objekterkennung, medizinische Bildanalyse
  - **NLP**: Sprachübersetzung, Chatbots, Sentiment Analysis
  - **Predictive Maintenance**: Sensordaten, Zeitreihen mit vielen Features
  - **Generative Tasks**: Bild-/Texterstellung, Data Augmentation
- **Beispiele für klassisches ML**:
  - **Tabellarische Daten**: Kreditscoring, Kundenklassifikation
  - **Einfache Klassifikation**: Mit wenigen, gut verstandenen Features
  - **Regression**: Preisvorhersage, wissenschaftliche Analysen
  - **Clustering**: Kundensegmentierung, Anomalieerkennung
- **Hybride Ansätze**:
  - **Feature-Extraktion**: Deep Learning für Features + klassisches ML für Klassifikation
  - **Ensemble**: Kombination verschiedener Ansätze
  - **Transfer Learning**: Pretrained Models + Domain-spezifische Anpassung
- **Praktischer Tipp**: Beginne mit einfachsten Methoden, steigere Komplexität schrittweise

Zusammenfassung: Was haben wir gelernt?

- **Die Macht tiefer Netze**: Sie können komplexe Funktionen durch hierarchische Feature-Extraktion lernen. Tiefe ist hierbei oft effizienter als Breite.

- **Spezialisierung ist der Schlüssel**:
  - **CNNs**: Nutzen lokale Strukturen und Translation Invariance für Bilder. Convolution spart Parameter und lernt Kanten, Formen, Objekte.
  - **RNNs/LSTMs**: Modellieren Sequenzen und Zeitabhängigkeiten durch internen Status (Memory).
  - **Autoencoders**: Lernen komprimierte Repräsentationen ohne Labels (Unsupervised).
  - **GANs**: Erzeugen neue Daten durch einen Wettbewerb zweier Netze.
- **Das Fundament**:
  - Alles basiert auf linearer Algebra (Matrizen), Calculus (Gradienten) und Optimierung (Backpropagation).
  - Training ist ein iterativer Prozess der Fehlerminimierung (Loss Landscape).

Ausblick und Zukunft des Deep Learning

- **Der Trend zu Foundation Models**:
  - Weg von "Ein Modell pro Aufgabe" hin zu "Ein Modell für alles".
  - Transfer Learning wird Standard (Pre-training auf riesigen Daten, Fine-tuning für den speziellen Fall).
- **Generative AI & LLMs**:
  - Transformers (Attention) haben RNNs fast vollständig abgelöst.
  - Modelle verstehen nicht nur Syntax, sondern auch Semantik und Kontext.
  - → Mehr dazu in der kommenden Vorlesung "Generative AI"!
- **Herausforderungen**:
  - Energieverbrauch und Modellgröße.
  - Erklärbarkeit (Black Box Problem) und Fairness/Bias.
  - Robustheit gegen Adversarial Attacks.

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


N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," The journal of machine learning research, vol. 15, no. 1, pp. 1929–1958, 2014.


A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in Advances in neural information processing systems, vol. 25, pp. 1097–1105, 2012.


K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," arXiv preprint arXiv:1409.1556, 2014.


R. K. Srivastava, K. Greff, and J. Schmidhuber, "Highway networks," arXiv preprint arXiv:1505.00387, 2015.


K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 770–778, 2016.


A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," arXiv preprint arXiv:1704.0486, 2017.


M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in International conference on machine learning, pp. 6105–6114, 2019.


A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, et al., "An image is worth 16x16 words: Transformers for image recognition at scale," arXiv preprint arXiv:2010.11929, 2020.

References III


J. L. Elman, “Finding structure in time,” Cognitive science, vol. 14, no. 2, pp. 179–211, 1990.


S. Hochreiter, “Untersuchungen zu dynamischen neuronalen netzen,” in Diploma thesis. Institut für Informatik, Technische Universität München, 1991.


Y. Bengio, P. Simard, and P. Frasconi, “Learning long-term dependencies with gradient descent is difficult,” IEEE transactions on neural networks, vol. 5, no. 2, pp. 157–166, 1994.


P. J. Werbos, “Backpropagation through time: what it does and how to do it,” Proceedings of the IEEE, vol. 78, no. 10, pp. 1550–1560, 1990.


J. Schmidhuber, “Deep learning in neural networks: An overview,” Neural networks, vol. 61, pp. 85–117, 2015.


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


A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” Advances in neural information processing systems, vol. 30, 2017.


J. Schmidhuber, “Learning to control fast-weight memories: An alternative to dynamic recurrent networks,” Neural Computation, vol. 4, no. 1, pp. 131–139, 1992.


R. M. Neal, Bayesian learning for neural networks, vol. 118.
Springer, 1996.

References IV


J. Schmidhuber, “Making the world differentiable: On using self-supervised fully recurrent neural networks for dynamic reinforcement learning and planning in non-stationary environments,” in Technical Report FKI-126-90, TU Munich, 1990.


I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative adversarial nets,” in Advances in neural information processing systems, vol. 27, 2014.


A. Radford, L. Metz, and S. Chintala, “Unsupervised representation learning with deep convolutional generative adversarial networks,” arXiv preprint arXiv:1511.06434, 2015.


M. Arjovsky, S. Chintala, and L. Bottou, “Wasserstein gan,” in International conference on machine learning, pp. 214–223, 2017.


M. Mirza and S. Osindero, “Conditional generative adversarial nets,” arXiv preprint arXiv:1411.1784, 2014.


T. Karras, S. Laine, and T. Aila, “A style-based generator architecture for generative adversarial networks,” in Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, pp. 4401–4410, 2019.