

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

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{pmatrix} \quad (1)$$

- **Zeilenvektor:** $\mathbf{x}^T = (x_1, x_2, \dots, x_d)$ (Transponiert)

- **Matrix:** $\mathbf{A} \in \mathbb{R}^{m \times n}$ mit m Zeilen und n Spalten

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

Mathematischer Refresher – Grundoperationen

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

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

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

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

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

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

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

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

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:** $\mathbf{y} = \alpha_1 \mathbf{x}_1 + \alpha_2 \mathbf{x}_2 + \dots + \alpha_k \mathbf{x}_k$

Mathematischer Refresher – Differentiation und Gradienten

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

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

- **Gradient:** Vektor aller partiellen Ableitungen

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

- **Kettenregel:** Für $f(g(x))$

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

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

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

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

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)) \quad (11)$$

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

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

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

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

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

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

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 $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ eine nicht-konstante, beschränkte und monotone Aktivierungsfunktion. Dann kann für jede stetige Funktion $g : [0, 1]^d \rightarrow \mathbb{R}$ und $\epsilon > 0$ eine Funktion

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

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:**

$$L(\theta + \Delta\theta) \approx L(\theta) + \nabla L(\theta)^T \Delta\theta \quad (16)$$

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

$$L(\theta - \eta \nabla L(\theta)) \approx L(\theta) - \eta \|\nabla L(\theta)\|^2 \leq L(\theta) \quad (17)$$

- **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

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

- mit dem Gewichtsvektor $\mathbf{w} \in \mathbb{R}^d$, Eingabevektor $\mathbf{x} \in \mathbb{R}^d$, Bias $b \in \mathbb{R}$ und Schwellwert θ

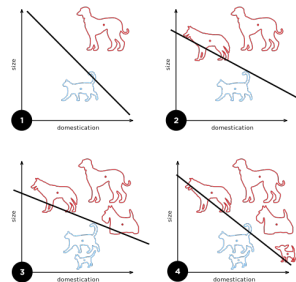
- Das Skalarprodukt: $\mathbf{w}^T \mathbf{x} = \sum_{i=1}^d w_i x_i$

- Entscheidungsgrenze im 2D-Fall ($d = 2$): Gerade mit Gleichung

$$w_1 x_1 + w_2 x_2 + b = \theta \quad (19)$$

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{b - \theta}{w_2} \quad (20)$$

- 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

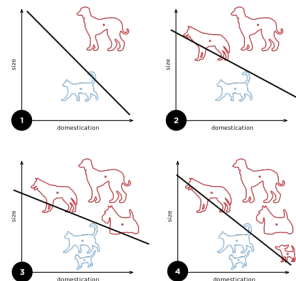
- Gegeben: Trainingsdatensatz $D = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ mit $\mathbf{x}_i \in \mathbb{R}^d, y_i \in \{-1, +1\}$
- Perceptron-Lernregel [3]:

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

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

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

- 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:

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

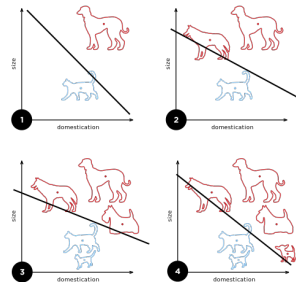
- $\eta > 0$: Lernrate (Schrittweite), θ : Parametervektor
- Gradient einer Funktion $f: \mathbb{R}^n \rightarrow \mathbb{R}$:

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

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

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

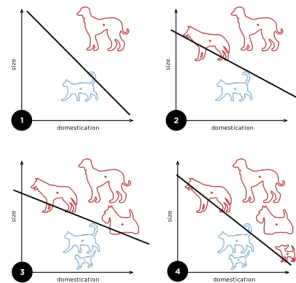
$$\nabla_{\mathbf{w}} L = - \sum_{(\mathbf{x}_i, y_i) \in M} y_i \mathbf{x}_i \quad (26)$$



Training von Neuronalen Netzen: Single-Layer-Perceptron

■ Daraus folgt:

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



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:

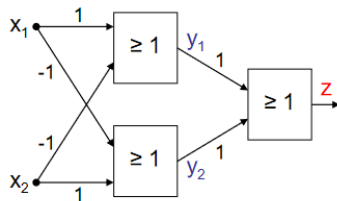
$$w_1 \cdot 0 + w_2 \cdot 1 + b > 0 \quad (\text{für } (0,1)) \quad (28)$$

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

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

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

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



Konstruktion von Neuronalen Netzen: Multi-Layer-Perceptron

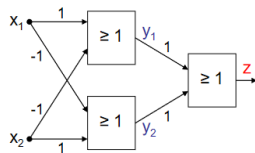
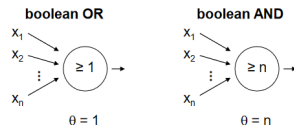
- **Lösung des XOR-Problems:** Mehrschichtige Netze!
- **Komposition von Hyperebenen:**
 - Erste Schicht: Erzeugt mehrere lineare Entscheidungsgrenzen
 - Zweite Schicht: Kombiniert diese zu komplexeren Formen
- **Mathematische Intuition für XOR:**

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

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

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

- **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:

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

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

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

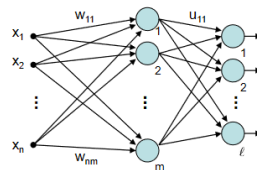
$$\hat{\mathbf{y}} = \sigma(\mathbf{z}^{(2)}) \quad (38)$$

- Mean Squared Error (MSE) Loss:

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

- 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

$$f(w) = \sum_{x \in B} ||g(w; x) - g^*(x)||^2 \rightarrow \min \quad (40)$$

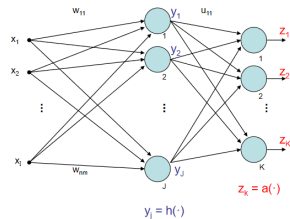
- mit dem Output des Netzes $g(w; x)$ und dem erwarteten Output $g^*(x)$

$$u^{(t+1)} = u^t - \gamma \nabla_u f(w_t, u_t) \quad (41)$$

$$w^{(t+1)} = w^t - \gamma \nabla_w f(w_t, u_t) \quad (42)$$

$$(43)$$

- 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:

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

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

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

- Rekursive Berechnung (rückwärts durch das Netz):

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

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

- Gradientenberechnung:

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

Training von Neuronalen Netzen: Multi-Layer-Perceptron

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

$$\begin{aligned}\nabla f(w, u) &= \sum_{x, z^* \in B} \nabla f(w, u; x, z^*) \\ \frac{\partial f(w, u)}{\partial u_{jk}} &= \sum_{x, z^* \in B} \frac{\partial f(w, u; x, z^*)}{\partial u_{jk}} \\ \frac{\partial f(w, u)}{\partial w_{ij}} &= \sum_{x, z^* \in B} \frac{\partial f(w, u; x, z^*)}{\partial w_{ij}}\end{aligned}$$

- mit den Sigmoid-Aktivierungsfunktionen:

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

- mit der Kettenregel für Ableitungen:

$$[p(q(x))]^\sim = p^\sim(q(x)) \cdot q^\sim(x)$$

- 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 :

$$\begin{aligned}
 f(w, u; x, z^*) &= \sum_{k=1}^K [a(u_k y) - z_k^*]^2 \\
 \frac{\partial f(w, u; x, z^*)}{\partial u_{jk}} &= \sum_{x, z^* \in B} \frac{\partial f(w, u; x, z^*)}{\partial u_{jk}} \\
 &= 2[a(u_k y) - z_k^*] \cdot a'(u_k y) \cdot y_j \\
 &= 2[a(u_k y) - z_k^*] \cdot a(u_k y) \cdot (1 - a(u_k y)) \cdot y_j \\
 &= 2 \underbrace{[z_k - z_k^*] \cdot z_k \cdot (1 - z_k)}_{\text{Fehlerterm } \delta_k} \cdot y_j \\
 \frac{\partial f(w, u; x, z^*)}{\partial w_{ij}} &= 2 \sum_{k=1}^K (a(u_k y) - z_k^*) \cdot a'(u_k y) \cdot u_{jk} \cdot h'(w_j x) x_i \\
 &= 2 \sum_{k=1}^K (z_k - z_k^*) \cdot z_k \cdot (1 - z_k) \cdot u_{jk} \cdot y_j (1 - y_j) x_i \\
 &= x_i y_j (1 - y_j) 2 \underbrace{\sum_{k=1}^K (z_k - z_k^*) \cdot z_k \cdot (1 - z_k) \cdot u_{jk} \cdot y_j}_{\text{Fehlerterm}}
 \end{aligned}$$

Verallgemeinerung Training von Neuronalen Netzen: M-Layer-Perceptron

- bei einem Neuronalen Netz mit L Layern S_1, S_2, \dots, 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:

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

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:

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

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

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

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

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

- 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

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

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

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

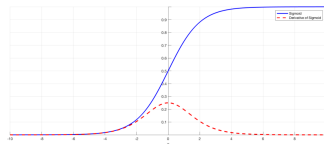
- **Intuition:**
 - Untere Schichten: Einfache Features (Kanten, Texturen)
 - Mittlere Schichten: Kombinationen (Formen, Teile)
 - Obere Schichten: Komplexe Konzepte (Objekte, Semantik)

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:

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

- 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)

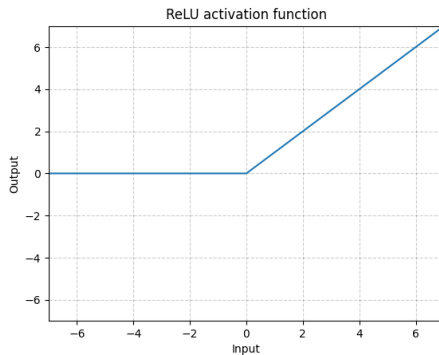
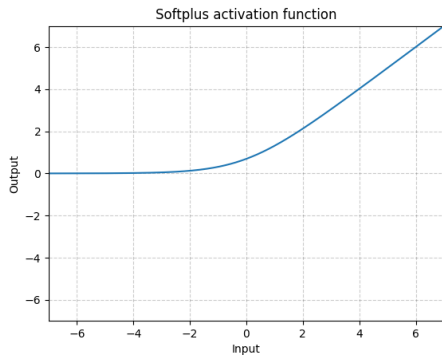


Aktivierungsfunktionen – Mathematische Eigenschaften und Warum sie wichtig sind

- **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 $|x| \gg 0$: $\sigma'(x) \rightarrow 0$
- **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:**

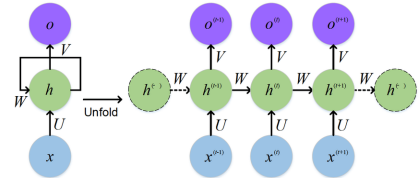
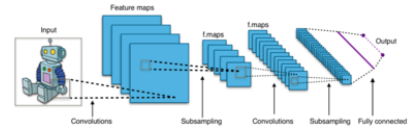
$$\text{Ohne } \sigma : f(\mathbf{x}) = \mathbf{W}_2(\mathbf{W}_1\mathbf{x}) = (\mathbf{W}_2\mathbf{W}_1)\mathbf{x} = \mathbf{W}_{\text{linear}}\mathbf{x} \quad (61)$$

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

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

■ 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}

$$\text{Conv}(\mathbf{T}_{\mathbf{t}}[\mathbf{x}]) = \mathbf{T}_{\mathbf{t}}[\text{Conv}(\mathbf{x})] \quad (63)$$

■ Pooling führt zu begrenzter Translationsinvarianz:

$$\text{MaxPool}(\mathbf{X})_{ij} = \max_{(p,q) \in N_{ij}} \mathbf{X}_{p,q} \quad (64)$$

■ Hierarchische Feature-Extraktion: Einfache \rightarrow Komplexe Features

Spezielle Netzwerkarchitekturen: CNN – Implementierung

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

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

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

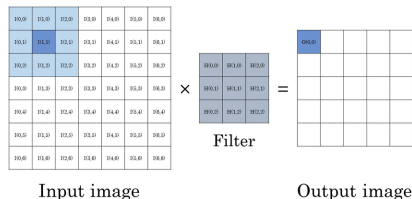
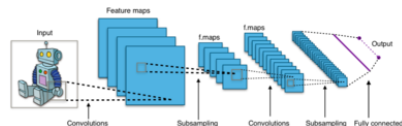
- **I**: Input-Feature-Map, **K**: Kernel (Filter) der Größe $M \times N$

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

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

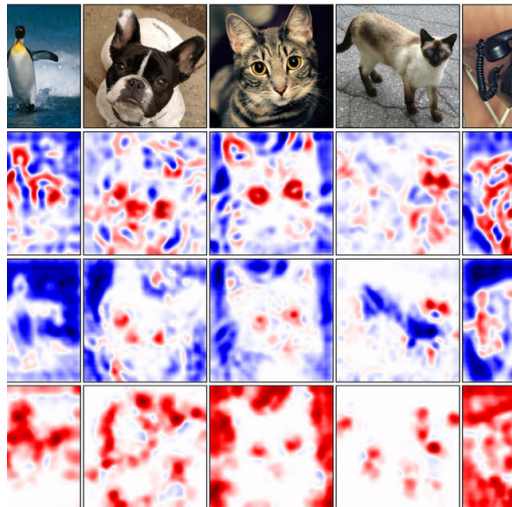
- **Parameter Sharing**: Derselbe Filter wird über gesamte Feature-Map angewendet

- **Translation Invariance**: Robustheit gegenüber Verschiebungen



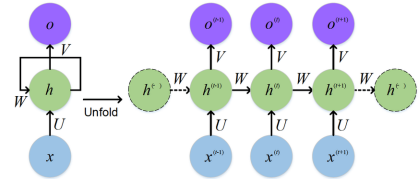
Spezielle Netzwerkarchitekturen: CNN

- Anschaulich: Formen werden erkannt
- Katzenohren sind anders als Hundeohren
- Verallgemeinerbar für andere Objektklassifizierungen



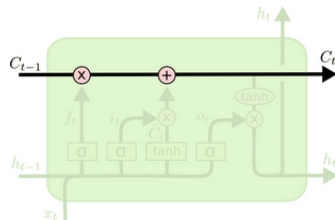
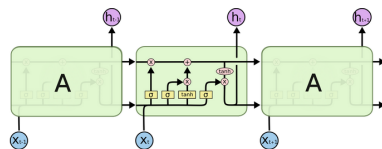
Spezielle Netzwerkarchitekturen: RNN

- Anschaulich: Schleife in Netzwerk "merkt" sich vorherige 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 C_t** : Langzeit-Gedächtnis der LSTM-Zelle
- **Hidden State 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 C_{t-1} gelöscht werden

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \quad (67)$$

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

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \quad (68)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \quad (69)$$

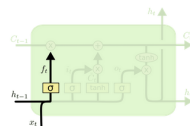
- **Cell State Update:**

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \quad (70)$$

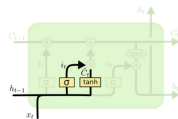
- **Output Gate und Hidden State:**

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \quad (71)$$

$$h_t = o_t * \tanh(C_t) \quad (72)$$



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

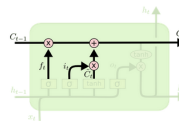


$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

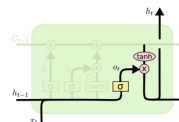
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

Spezielle Netzwerkarchitekturen: LSTM

- Update des alten Cell states mithilfe des alten und neuen Zustandes
- Output der LSTM Zelle und Erzeugung des neuen hidden states



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

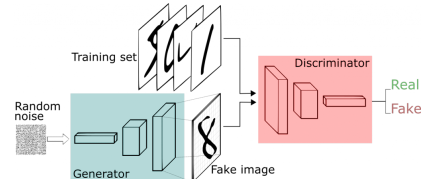
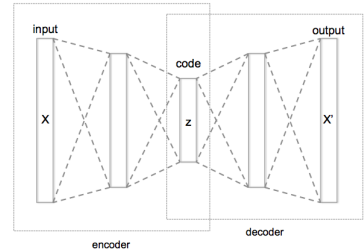


$$o_t = \sigma(W_o [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * \tanh(C_t)$$

Spezielle Netzwerkarchitekturen: Unsupervised Learning

- Autoencoder bzw. Encoder-Decoder Netzwerke
- Versuch den Input wieder zu rekonstruieren → Interessant für unsupervised anomaly detection
- Generative Netzwerk
- Möglichkeit aus "Noise" komplexe echt aussehende Daten zu erzeugen → mögliche Beschleunigung von (FE-)Simulationen



Was ist Predictive Maintenance?

Grundlegende Ziele

- Prozessüberwachung und eventuelle Steuerung
- Vorhersagen von Maschinen/Produktionsausfällen
- Hilfe/Unterstützung bei der Wartungsplanung
- Klassifikation von Fehlerzuständen
- ...

Was ist Predictive Maintenance?

- PM ist als Teilbereich der Industrie 4.0 zu verstehen
- (Nah-) Echtzeitdatenanalysen sollen Bedarfsgerechte Wartung ermöglichen
- Großes Potential der Kosteneinsparung

Fallbeispiel: Vollautomatisierte Produktionszelle

- Automatisierungstechnik liefert Prozessdaten
- zusätzliche Sensorik liefert mehr Daten
- Ein Trend in den Daten deutet auf einen zukünftigen Fehler hin
- Der Fehler wird klassifiziert und eine Handlungsanweisung wird herausgegeben
- Der Fehler kann, bevor er größere Schäden, oder Produktionsausfälle behoben werden
- Welche Schritte sind dafür Notwendig?

Fallbeispiel: Vollautomatisierte Produktionszelle – Outline

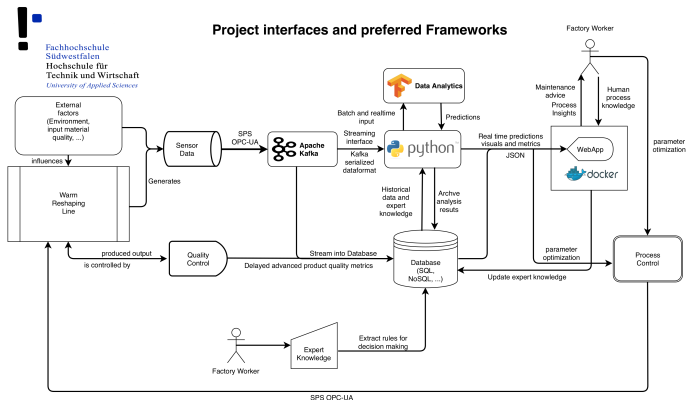
Ausgangslage Industrie 3.0

- Automatisierungstechnik existiert und Daten stehen der Anlagensteuerung zur Verfügung
- Wie werden die Daten weiterverarbeitet? Haben Sie Beispiele aus Ihrem Unternehmen?
- eine Möglichkeit: Auslesen der Daten über eine OPC-UA Schnittstelle
- Weiterleitung dieser Daten über ein schnelle und skalierbare Schnittstelle (MQTT, ApacheKafka, ...)
- Empfang dieser Daten auf einem Datenbankserver (MariaDB, TimeseriesDB, ApacheDruid, ...)
- Durchführung von Echtzeitanalysen auf einem Edge-Computer, oder in der Cloud

Fallbeispiel: Vollautomatisierte Produktionszelle – Warmumformanlage

- Umformung und gleichzeitige Härtung von Stahl im Automobilleichtbau
- Probleme:
 - Komplexe Wirkzusammenhänge während des Prozesses
 - Verlust von Know-How bei Standortwechseln der Produktion
 - Langwierige Prüfverfahren zur Qualitätssicherung → hohes Schadenspotential bei unentdeckten Fehlern

Visualisierung eines möglichen Backends



Fallbeispiel: Vollautomatisierte Produktionszelle – Datensammlung

- Prozessdaten werden mit einer OPC-UA Schnittstelle aus der übergeordneten Steuerung gelesen
- Die Qualitätssicherung liefert verzögert Daten über die gefertigten Bauteile
- Experten geben Know-How über den Prozess in einer strukturierten Form einer FMEA (Failure Mode and Effects Analysis) ein

Fallbeispiel: Vollautomatisierte Produktionszelle – Erste Datenverarbeitung mit einem Broker System

- Die Daten aus der Anlagensteuerung werden mit einem "Producer" an einen Broker geschickt, der den Datenstrom verwaltet
- Ein Brokersystem wie ApacheKafka oder MQTT kontrolliert den Datenfluss
- Clients können dem Broker "folgen" (subscriben)
- Ein sogenannter Consumer erhält diesen nun bei jedem neuen erzeugten Datenpunkt
- Diese Datenpunkte können dann mit einem Machine Learning Framework (sci-kit-learn, TensorFlow, PyTorch) analysiert werden
- Vorteil: Parallelisierung des Datenflusses möglich
- Gleichzeitiges Speichern der Daten in einer Datenbank und Echtzeitverarbeitung im Datenanalyse Framework und darstellung auf

Fallbeispiel: Vollautomatisierte Produktionszelle – Darstellung der Daten für einen Maschinenoperator

- Eine vereinfachte visuelle Darstellung der Maschinenparameter kann z.B. in einem Dashboard ausgegeben werden
- Erfahrene Operatoren können aus diesen Daten und ersten Analysen Schlussfolgerungen ziehen
- mögliche Parameteranpassungen im Prozess können durch diesen Operator durchgeführt werden

Fallbeispiel: Vollautomatisierte Produktionszelle – Direkte Handlungsanweisungen an einen Operator

- Vorhersagen des trainierten Machine Learning Modells werden mit einer FMEA abgeglichen
- eine FMEA enthält die häufigsten Fehler und Defekte einer Anlage
- Problemlösungen werden hier strukturiert skizziert
- unerfahrene Operatoren können hier durch das gesammelte Expertenwissen Entscheidungen treffen
- Operatoren können für den Vorhergesagten Zeitpunkt Mechaniker zur Wartung der Maschine anfordern

Fallbeispiel: Vollautomatisierte Produktionszelle – Direkte Handlungsanweisungen an die Prozesssteuerung (Dangerzone)

- Mögliche kritische Zustände können direkt zu einem Stop der Produktion führen
- Anpassung der Prozessparameter direkt durch die Software
→ vollständig autonomes System
- Mensch wird nur noch zum Eintragen von Know-How benötigt

Andere Beispiele: Wo ist PM finanziell besonders sinnvoll?

- Kraftfahrzeuge: Sensorik in Verschleißteilen kann Totalausfälle vermeiden
- Luftfahrt: Der Ausfall von Passagier- oder Luftfrachtflügen kann mit guten Ausfallvorhersagen verhindert werden
- Schienenverkehr

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.