



Deep Learning

Einführung - Thema 2

Silas Hoffmann

11. April 2020

Fachhochschule Wedel

Deep Learning

2020-04-11



Deep Learning

Einführung - Thema 2

Silas Hoffmann
11. April 2020
Fachhochschule Wedel

Geschichtliche Entwicklung

McCulloch-Pitts-Neuron

Perceptron

Adeline

Convolutional Neural Network

Aktuelle Entwicklung

Backpropagation

Multilayer Perceptron

Recurrent Neural Network

2020-04-11

Deep Learning

└─ Geschichtliche Entwicklung

└─ McCulloch-Pitts-Neuron

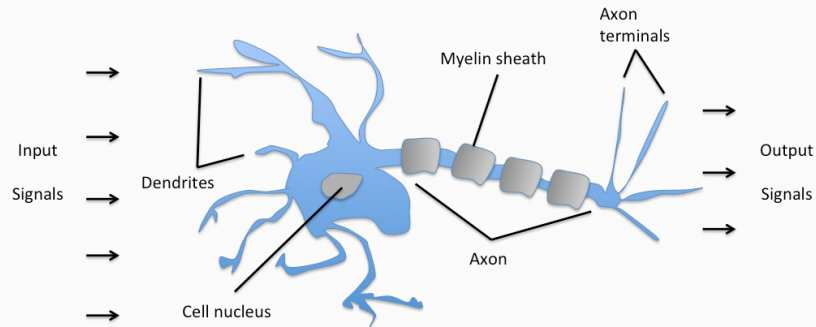
Geschichtliche Entwicklung

McCulloch-Pitts-Neuron

Geschichtliche Entwicklung

McCulloch-Pitts-Neuron

Zusammenhang - Biologisches Neuron



Schematic of a biological neuron.

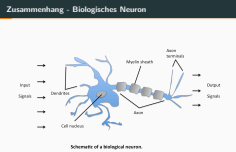
2020-04-11

Deep Learning

└─ Geschichtliche Entwicklung

└─ McCulloch-Pitts-Neuron

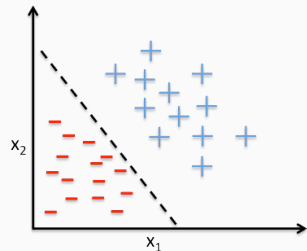
└─ Zusammenhang - Biologisches Neuron



This is small

1. *Dendriten*: Nehmen Infos auf
2. This is small
 - This is large

- Modell soll Funktionalität des biologischen Neurons imitieren
- Klassifizierungsproblem als grundlegende Problemstellung
- Lineare Entscheidungsfunktion zur binären Klassifizierung verwendet



Example of a linear decision boundary for binary classification.

2020-04-11

Deep Learning

└─ Geschichtliche Entwicklung

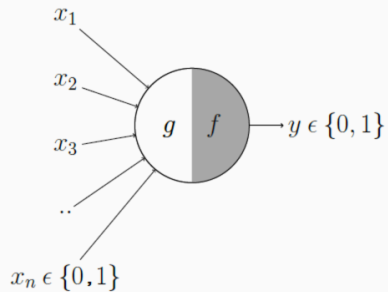
└─ McCulloch-Pitts-Neuron

└─ MP-Neuron

- Modell soll Funktionalität des biologischen Neurons imitieren
- Klassifizierungsproblem als grundlegende Problemstellung
- Lineare Entscheidungsfunktion zur binären Klassifizierung verwendet



Aufbau und Funktionsweise



$$g(x_1, x_2, \dots, x_n) = g(x) = \sum_{i=1}^n x_i$$

$$f(g(x)) = \begin{cases} 1 & \text{if } g(x) \geq \theta \\ 0 & \text{if } g(x) < \theta \end{cases}$$

2020-04-11

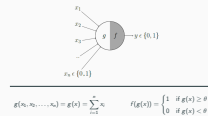
Deep Learning

└─ Geschichtliche Entwicklung

└─ McCulloch-Pitts-Neuron

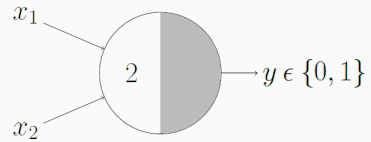
└─ Aufbau und Funktionsweise

Aufbau und Funktionsweise



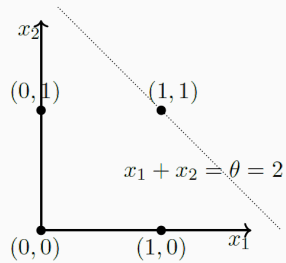
$$g(x_1, x_2, \dots, x_n) = g(x) = \sum_{i=1}^n x_i \quad f(g(x)) = \begin{cases} 1 & \text{if } g(x) \geq \theta \\ 0 & \text{if } g(x) < \theta \end{cases}$$

Notation AND-Gatter



AND function

$$x_1 + x_2 = \sum_{i=1}^2 x_i \geq 2$$



2020-04-11

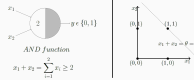
Deep Learning

└─ Geschichtliche Entwicklung

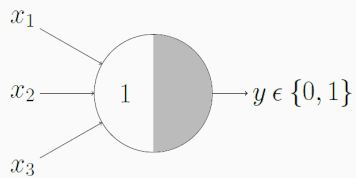
└─ McCulloch-Pitts-Neuron

└─ Notation AND-Gatter

Notation AND-Gatter

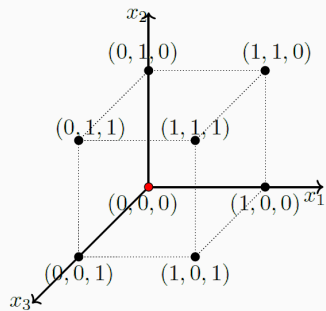


Notation OR-Gatter



OR function

$$x_1 + x_2 + x_3 = \sum_{i=1}^3 x_i \geq 1$$



2020-04-11

Deep Learning

└─ Geschichtliche Entwicklung

└─ McCulloch-Pitts-Neuron

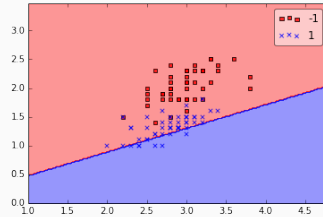
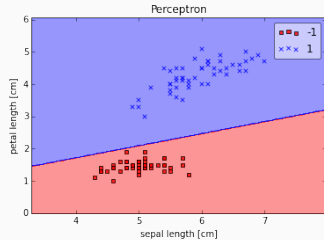
└─ Notation OR-Gatter

Notation OR-Gatter



Nachteile

- Keine kontinuierlichen Eingabewerte (nur boolesche Werte)
- Schwelle muss manuell gesetzt werden, keine automatische Aktualisierung vorgesehen
- Keine Priorisierungsmöglichkeit der Eingabewerte möglich
- Funktionen müssen durch lineare Entscheidungsfunktion getrennt werden können



2020-04-11

Deep Learning

- └ Geschichtliche Entwicklung
 - └ McCulloch-Pitts-Neuron
 - └ Nachteile

Nachteile

- Keine kontinuierlichen Eingabewerte (nur boolesche Werte)
- Schwelle muss manuell gesetzt werden, keine automatische Aktualisierung vorgesehen
- Keine Priorisierungsmöglichkeit der Eingabewerte möglich
- Funktionen müssen durch lineare Entscheidungsfunktion getrennt werden können



2020-04-11

Deep Learning
└─ Geschichtliche Entwicklung
 └─ Perceptron

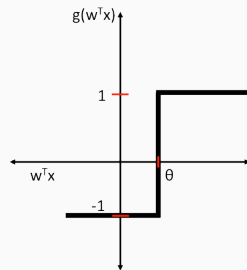
Geschichtliche Entwicklung
Perceptron

Geschichtliche Entwicklung

Perceptron

Perceptron

- Ähnliche Aktivierungsfunktion wie beim MP-Neuron
- Jedoch gewichtete kontinuierliche Eingabewerte



Unit step function.

$$\mathbf{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$$

$$\begin{aligned} z &= w_1 x_1 + \dots + w_m x_m \\ &= \sum_{j=1}^m x_j w_j \\ &= \mathbf{w}^T \mathbf{x} \end{aligned}$$

2020-04-11

Deep Learning

└─ Geschichtliche Entwicklung

└─ Perceptron

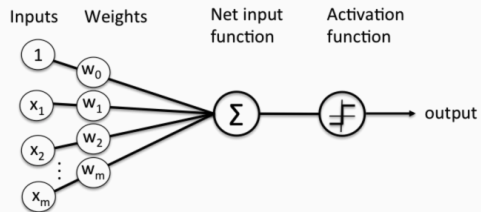
└─ Perceptron

Perceptron

- Ähnliche Aktivierungsfunktion wie beim MP-Neuron
- Jedoch gewichtete kontinuierliche Eingabewerte



$$\begin{aligned} \mathbf{w} &= \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \\ z &= w_1 x_1 + \dots + w_m x_m \\ &= \sum_{j=1}^m x_j w_j \\ &= \mathbf{w}^T \mathbf{x} \end{aligned}$$



Schematic of Rosenblatt's perceptron.

$$g(z) = \begin{cases} 0 & \text{if } z \leq 0 \\ 1 & \text{if } z > 0 \end{cases}$$

$$\begin{aligned} z &= \mathbf{w_0x_0} + w_1x_1 + \dots + w_mx_m \\ &= \sum_{j=0}^m x_j w_j \\ &= \mathbf{w^T x} \end{aligned}$$

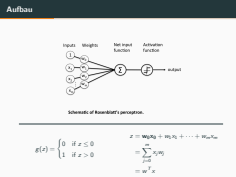
2020-04-11

Deep Learning

└ Geschichtliche Entwicklung

└ Perceptron

└ Aufbau



2020-04-11

Deep Learning

└─ Geschichtliche Entwicklung

└─ Perceptron

└─ Lernregel - Ablauf

- Modell übernimmt selbst die Anpassung der Gewichte
- Test mittels einer Menge von gelabelten Trainingsdatensätzen

Grober Ablauf

- Initialisiere die Gewichte mit einem sehr kleinen Wert oder 0.
- Für jeden Datensatz der Menge von Trainingsdatensätzen:
 - Berechne den Ausgabewert des Systems
 - Gleiche die Gewichte an

- Modell übernimmt selbst die Anpassung der Gewichte
- Test mittels einer Menge von gelabelten Trainingsdatensätzen

Grober Ablauf

- Initialisiere die Gewichte mit einem sehr kleinen Wert oder 0.
- Für jeden Datensatz der Menge von Trainingsdatensätzen:
 - Berechne den Ausgabewert des Systems
 - Gleiche die Gewichte an

Angleichung der Gewichte

- Gewichte komponentenweise angleichen: $w_j := w_j + \Delta w_j$
- Gewichtsänderung: $\Delta w_j = \eta (\text{target}^{(i)} - \text{output}^{(i)}) x_j^{(i)}$
- Beispiel - Iteration mit zweidimensionalem Trainingsvektor:

$$\Delta w_0 = \eta (\text{target}^{(i)} - \text{output}^{(i)})$$

$$\Delta w_1 = \eta (\text{target}^{(i)} - \text{output}^{(i)}) x_1^{(i)}$$

$$\Delta w_2 = \eta (\text{target}^{(i)} - \text{output}^{(i)}) x_2^{(i)}$$

2020-04-11

Deep Learning

└─ Geschichtliche Entwicklung

└─ Perceptron

└─ Lernregel - Formel

Angleichung der Gewichte

- Gewichte komponentenweise angleichen: $w_j := w_j + \Delta w_j$
- Gewichtsänderung: $\Delta w_j = \eta (\text{target}^{(i)} - \text{output}^{(i)}) x_j^{(i)}$

- Beispiel - Iteration mit zweidimensionalem Trainingsvektor:

$$\Delta w_0 = \eta (\text{target}^{(i)} - \text{output}^{(i)})$$

$$\Delta w_1 = \eta (\text{target}^{(i)} - \text{output}^{(i)}) x_1^{(i)}$$

$$\Delta w_2 = \eta (\text{target}^{(i)} - \text{output}^{(i)}) x_2^{(i)}$$

- Trainingsdatensatz richtig erkannt:

$$\Delta w_j = \eta(1^{(i)} - 1^{(i)}) x_j^{(i)} = 0$$

$$\Delta w_j = \eta(1^{(i)} - 1^{(i)}) x_j^{(i)} = 0$$

- Trainingsdatensatz falsch erkannt:

$$\Delta w_j = \eta(1^{(i)} - -1^{(i)}) x_j^{(i)} = \eta(2) x_j^{(i)}$$

$$\Delta w_j = \eta(-1^{(i)} - 1^{(i)}) x_j^{(i)} = \eta(-2) x_j^{(i)}$$

- Trainingsdatensatz richtig erkannt:

$$\Delta w_j = \eta(1^{(i)} - 1^{(i)}) x_j^{(i)} = 0$$

$$\Delta w_j = \eta(1^{(i)} - 1^{(i)}) x_j^{(i)} = 0$$

- Trainingsdatensatz falsch erkannt:

$$\Delta w_j = \eta(1^{(i)} - -1^{(i)}) x_j^{(i)} = \eta(2) x_j^{(i)}$$

$$\Delta w_j = \eta(-1^{(i)} - 1^{(i)}) x_j^{(i)} = \eta(-2) x_j^{(i)}$$

2020-04-11

Deep Learning

└─ Geschichtliche Entwicklung

└─ Adeline

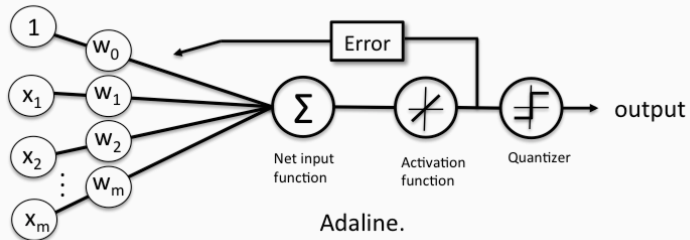
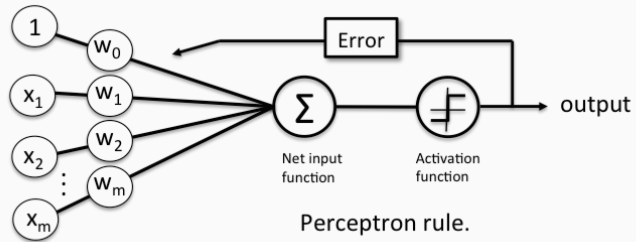
Geschichtliche Entwicklung

Adeline

Geschichtliche Entwicklung

Adeline

ADaptive LINear Element



2020-04-11

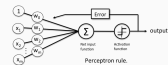
Deep Learning

└─ Geschichtliche Entwicklung

└─ Adeline

└─ ADaptive LINear Element

ADaptive LINear Element



- Lernalgorithmus durch Erfinder geprägt
- auch unter *Least-Mean-Square-Algorithmus* bekannt
- Wesentlicher Vorteil: Ableitbare Kostenfunktion

Notation

$$J(w) = \frac{1}{2} \sum_i (\text{target}^{(i)} - \text{output}^{(i)})^2 \quad \text{output}^{(i)} \in \mathbb{R}$$

2020-04-11

Deep Learning

└─ Geschichtliche Entwicklung

└─ Adeline

└─ Delta-Regel

- Lernalgorithmus durch Erfinder geprägt
- auch unter *Least-Mean-Square-Algorithmus* bekannt
- Wesentlicher Vorteil: Ableitbare Kostenfunktion

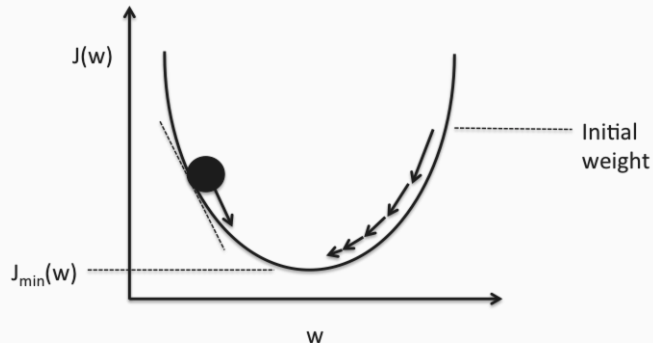
Notation

$$J(w) = \frac{1}{2} \sum_i (\text{target}^{(i)} - \text{output}^{(i)})^2 \quad \text{output}^{(i)} \in \mathbb{R}$$

Gradientenverfahren

- Ziel: Gradientenvektor für bestimmten Input bestimmen:

$$\nabla J \equiv \left(\frac{\partial J}{\partial w_1}, \dots, \frac{\partial J}{\partial w_m} \right)^T.$$



Schematic of gradient descent.

2020-04-11

Deep Learning

└ Geschichtliche Entwicklung

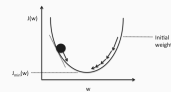
└└ Adeline

└└└ Gradientenverfahren

Gradientenverfahren

- Ziel: Gradientenvektor für bestimmten Input bestimmen:

$$\nabla J \equiv \left(\frac{\partial J}{\partial w_1}, \dots, \frac{\partial J}{\partial w_m} \right)^T.$$



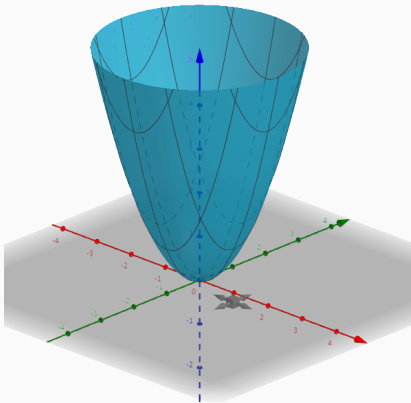
Schematic of gradient descent.

Partielle Ableitungen

- Differenzieren von Funktionen mit mehreren Eingabewerten
- Beispiel: $z = f(x) = x^2 + y^2$

Partielle Ableitung - Notation

$$\frac{\partial \text{Abzuleitende Fkt.}}{\partial \text{Betrachtete Komponente}}$$



2020-04-11

Deep Learning

└ Geschichtliche Entwicklung

└ Adeline

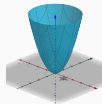
└ Partielle Ableitungen

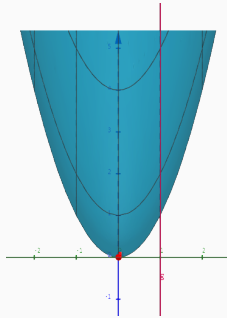
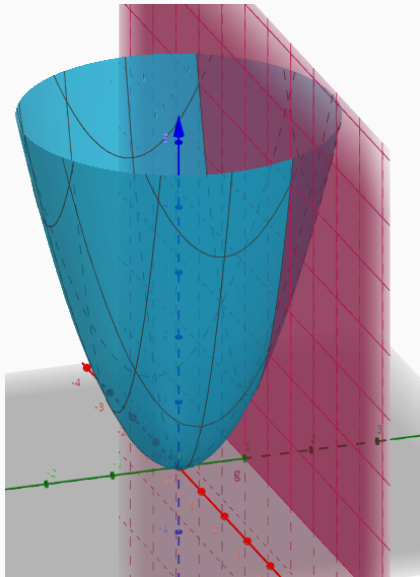
Partielle Ableitungen

- Differenzieren von Funktionen mit mehreren Eingabewerten
- Beispiel: $z = f(x) = x^2 + y^2$

Partielle Ableitung - Notation

$$\frac{\partial \text{Abzuleitende Fkt.}}{\partial \text{Betrachtete Komponente}}$$





Ableitung - Beispiel

$$z = f(x, y) = x^2 + y^2$$

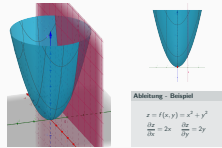
$$\frac{\partial z}{\partial x} = 2x \quad \frac{\partial z}{\partial y} = 2y$$

2020-04-11

Deep Learning

└ Geschichtliche Entwicklung

└ Adeline

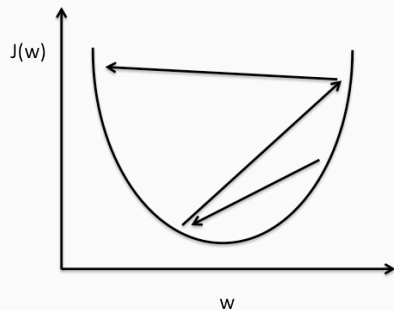


Ableitung - Beispiel

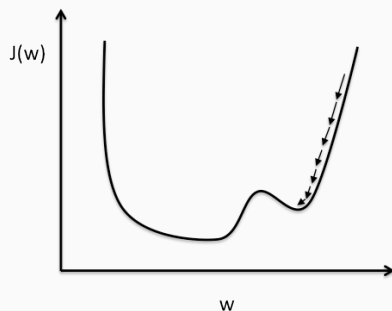
$$z = f(x, y) = x^2 + y^2$$

$$\frac{\partial z}{\partial x} = 2x \quad \frac{\partial z}{\partial y} = 2y$$

Gradientenverfahren



Large learning rate: Overshooting.



Small learning rate: Many iterations until convergence and trapping in local minima.

2020-04-11

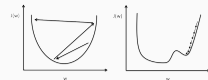
Deep Learning

└ Geschichtliche Entwicklung

└└ Adeline

└└└ Gradientenverfahren

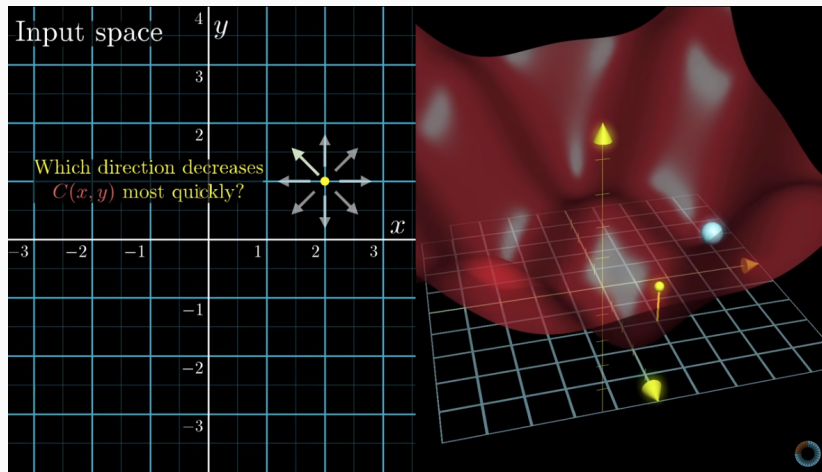
Gradientenverfahren



Large learning rate: Overshooting.

Small learning rate: Many iterations until convergence and trapping in local minima.

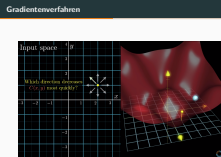
Gradientenverfahren



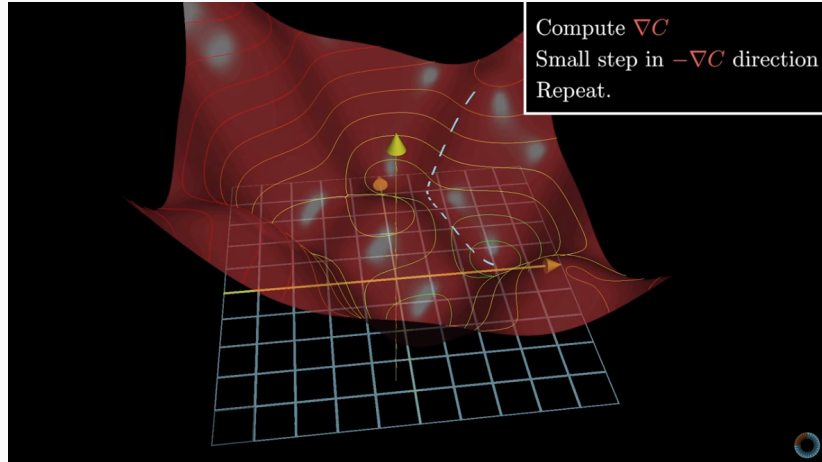
2020-04-11

Deep Learning

- └ Geschichtliche Entwicklung
 - └ Adeline
 - └ Gradientenverfahren



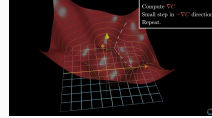
Gradientenverfahren



2020-04-11

Deep Learning
└─ Geschichtliche Entwicklung
 └─ Adeline
 └─ Gradientenverfahren

Gradientenverfahren



Gradientenverfahren - Anwendung

- Gradientenvektor

$$\nabla J \equiv \left(\frac{\partial J}{\partial w_1}, \dots, \frac{\partial J}{\partial w_m} \right)^T.$$

- Allgemein: Vektorielle Darstellung

$$\Delta w = -\eta \nabla J(w)$$

- Für die jeweiligen Gewichte: Komponentenweise Darstellung

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j}$$

- Angleichung der Gewichte $w = w + \Delta w$

2020-04-11

Deep Learning

└─ Geschichtliche Entwicklung

└─ Adeline

└─ Gradientenverfahren

Gradientenverfahren - Anwendung

- Gradientenvektor

$$\nabla J \equiv \left(\frac{\partial J}{\partial w_1}, \dots, \frac{\partial J}{\partial w_m} \right)^T.$$

- Allgemein: Vektorielle Darstellung

$$\Delta w = -\eta \nabla J(w)$$

- Für die jeweiligen Gewichte: Komponentenweise Darstellung

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j}$$

- Angleichung der Gewichte $w = w + \Delta w$

Kostenfunktion ableiten

$$\begin{aligned}\frac{\partial J}{\partial w_j} &= \frac{\partial}{\partial w_j} \frac{1}{2} \sum_i (t^{(i)} - o^{(i)})^2 \\ &= \frac{1}{2} \sum_i \frac{\partial}{\partial w_j} (t^{(i)} - o^{(i)})^2 \\ &= \frac{1}{2} \sum_i 2(t^{(i)} - o^{(i)}) \frac{\partial}{\partial w_j} (t^{(i)} - o^{(i)}) \\ &= \sum_i (t^{(i)} - o^{(i)}) \frac{\partial}{\partial w_j} \left(t^{(i)} - \sum_j w_j x_j^{(i)} \right) \\ &= \sum_i (t^{(i)} - o^{(i)}) (-x_j^{(i)})\end{aligned}$$

2020-04-11

Deep Learning

└─ Geschichtliche Entwicklung

└─ Adeline

└─ Kostenfunktion ableiten

Kostenfunktion ableiten

$$\begin{aligned}\frac{\partial J}{\partial w_j} &= \frac{\partial}{\partial w_j} \frac{1}{2} \sum_i (t^{(i)} - o^{(i)})^2 \\ &= \frac{1}{2} \sum_i \frac{\partial}{\partial w_j} (t^{(i)} - o^{(i)})^2 \\ &= \frac{1}{2} \sum_i 2(t^{(i)} - o^{(i)}) \frac{\partial}{\partial w_j} (t^{(i)} - o^{(i)}) \\ &= \sum_i (t^{(i)} - o^{(i)}) \frac{\partial}{\partial w_j} \left(t^{(i)} - \sum_j w_j x_j^{(i)} \right) \\ &= \sum_i (t^{(i)} - o^{(i)}) (-x_j^{(i)})\end{aligned}$$

Geschichtliche Entwicklung

Convolutional Neural Network

2020-04-11

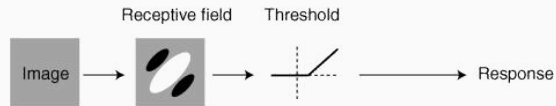
Deep Learning

└─ Geschichtliche Entwicklung

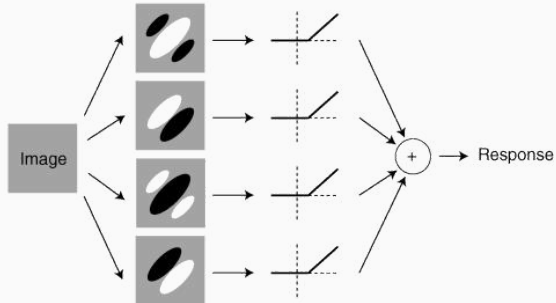
└─ Convolutional Neural Network

Geschichtliche Entwicklung
Convolutional Neural Network

A Simple cell



B Complex cell



2020-04-11

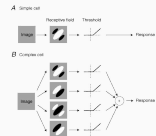
Deep Learning

└ Geschichtliche Entwicklung

└ Convolutional Neural Network

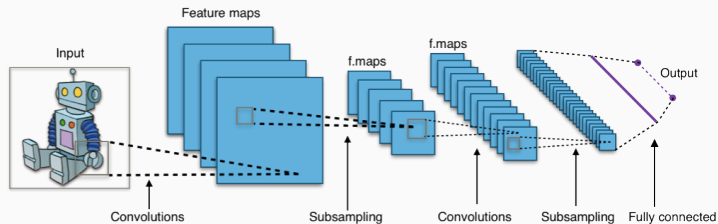
└ Biologische Zellarten

Biologische Zellarten



1. 1962: zwei Neurophysiologen Torsten Wiesel und David Hubel
2. Konzept der simple und complex cells
3. nicht positionsbunden - spatial invariance, räumliche Invarianz
4. Arten von Zellen zur Erkennung einfacher Kanten und Balken
5. *simple cells*: ist Positionsgebunden
6. *complex cells*: Muster können an beliebigen Positionen auftauchen
7. 1962: Konzept wie im Bild
8. 1980er Dr. Kunihiko Fukushima: erstes Modell nach diesem Konzept

- Yann LeCun: erstes Modell zum Erkennen von Handschrift
- *Verwendung von MNIST database of handwritten digits*
 - 60.000 Trainingsdatensätze
 - 10.000 zum Berechnen des Fehlers



2020-04-11

Deep Learning

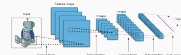
Geschichtliche Entwicklung

Convolutional Neural Network

Anfänge

Anfänge

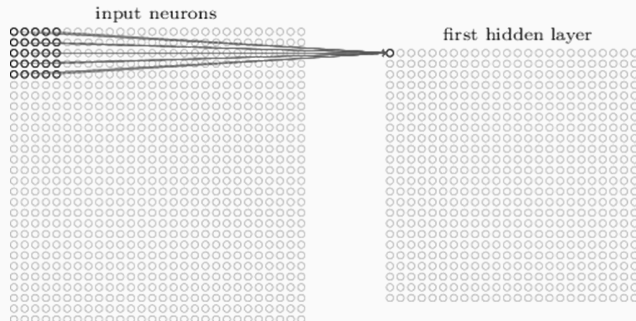
- Yann LeCun: erstes Modell zum Erkennen von Handschrift
- Verwendung von *MNIST database of handwritten digits*
 - 60.000 Trainingsdatensätze
 - 10.000 zum Berechnen des Fehlers



1. Pioniere, fr. Informatiker Yann LeCun
2. Bekannteste Ausarbeitung über CNN für Handschriften

Convolutinal Layer - Filter

- Mehrdimensionales Array mit Farbwerten zur Repräsentation im Rechner
- Durch Filter auf bestimmte *Low-Level* Eigenschaften schließen



Visualization of 5 x 5 filter convolving around an input volume and producing an activation map

2020-04-11

Deep Learning

Geschichtliche Entwicklung

Convolutional Neural Network

Convolutinal Layer - Filter

Convolutinal Layer - Filter

- Mehrdimensionales Array mit Farbwerten zur Repräsentation im Rechner
- Durch Filter auf bestimmte *Low-Level* Eigenschaften schließen



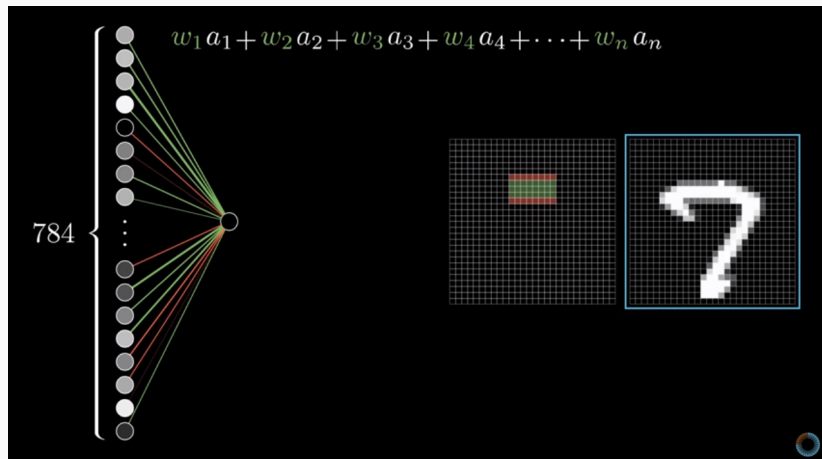
1. Farbwertearray kann pro Pixel mehrere Werte enthalten

- Generell
 - Besitzt feste Pixelgröße (*Kernelsize*) & Schrittweite
 - Scannt Bild Zeilenweise
 - *Padding* legt Verfahren für Rand des Bildes fest
 - Ausgabe wird *activation* oder *feature map* genannt
- Praxis
 - Convolutional Layer mit 32 oder 16 Bit
 - Jeder Filter generiert eigene Ausgabematrix
 - Nächster Convolutional Layer verwendet Ausgabematrizen als Input
 - Ausgabe wird in *Pooling Layer* gesteckt

- Generell
 - Besitzt feste Pixelgröße (*Kernelsize*) & Schrittweite
 - Scannt Bild Zeilenweise
 - *Padding* legt Verfahren für Rand des Bildes fest
 - Ausgabe wird *activation* oder *feature map* genannt
- Praxis
 - *Convolutional Layer* mit 32 oder 16 Bit
 - Jeder Filter generiert eigene Ausgabematrix
 - Nächster Convolutional Layer verwendet Ausgabematrizen als Input
 - Ausgabe wird in *Pooling Layer* gesteckt

1. Bsp. Filter 2×2 , Schrittweite: 2 - führt zu Halbierung der InputMatrix
2. Im Bsp. hängen immer 4 Pixel an einem Filter, die Eingabematrix wird gefaltet (convolute)

Filter - Funktionsweise



2020-04-11

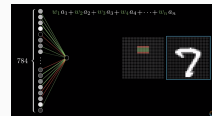
Deep Learning

└ Geschichtliche Entwicklung

└ Convolutional Neural Network

└ Filter - Funktionsweise

Filter - Funktionsweise



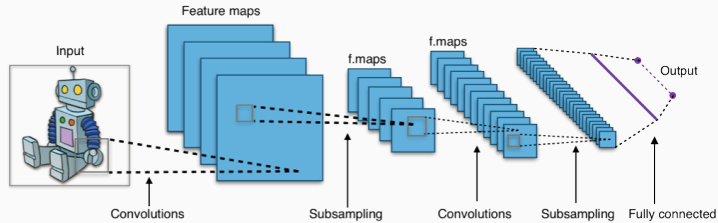
- Aggregiert die Ergebnisse von Convolutional Layern
- Ziele
 - Nur die relevantesten Signale an nächste Schicht weitergeben
 - Anzahl der Parameter im Netz reduzieren
- MaxPooling Layer am weitesten verbreitet

- Aggregiert die Ergebnisse von Convolutional Layern
- Ziele
 - Nur die relevantesten Signale an nächste Schicht weitergeben
 - Anzahl der Parameter im Netz reduzieren
- *MaxPooling Layer* am weitesten verbreitet

1. während die Größe des Inputs durch die Faltungen und das Pooling immer weiter reduziert wird, erhöht sich die Anzahl der Filter zur Erkennung von übergeordneten Signalen zunehmend

Fully Connected Layer

- Ausgangspunkt: *High-Level* Merkmale bereits durch frühere Schichten erkannt.
- Alle Neuronen der Ausgangsschicht sowie dieser Merkmale alle direkt miteinander verbunden
- Ausgabe sollte mit den richtigen Gewichten / Schwellwerten relativ eindeutige Ausgaben generieren



2020-04-11

Deep Learning

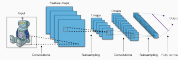
└ Geschichtliche Entwicklung

└ Convolutional Neural Network

└ Fully Connected Layer

Fully Connected Layer

- Ausgangspunkt: *High-Level* Merkmale bereits durch frühere Schichten erkannt.
- Alle Neuronen der Ausgangsschicht sowie dieser Merkmale alle direkt miteinander verbunden
- Ausgabe sollte mit den richtigen Gewichten / Schwellwerten relativ eindeutige Ausgaben generieren



Aktuelle Entwicklung

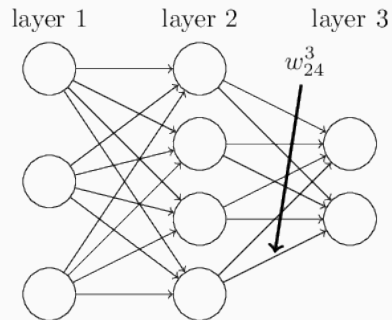
Backpropagation

2020-04-11

Deep Learning
└─ Aktuelle Entwicklung
 └─ Backpropagation

Aktuelle Entwicklung

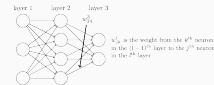
Backpropagation



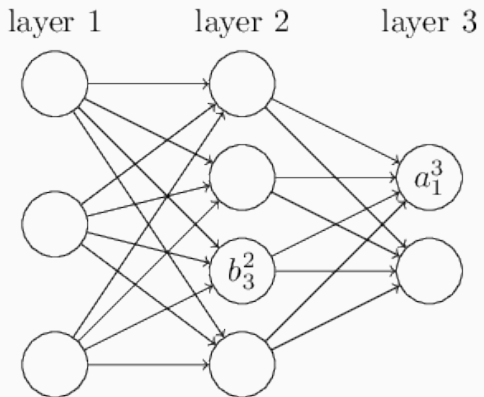
w_{jk}^l is the weight from the k^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer

2020-04-11

Deep Learning
└ Aktuelle Entwicklung
└ Backpropagation
└ Notation



1. l: Exponent, steht für die Schicht
2. l - 1, weil man stets von hinten nach vorne schaut
3. Eingabe wird auch als eigene Schicht verstanden
4. j: Index Zielneuron
5. k: Index Startneuron



$$a_j^l = \sigma \left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l \right) \Rightarrow \begin{aligned} a^l &= \sigma(z^l) \\ z^l &= w^l a^{l-1} + b^l \end{aligned}$$

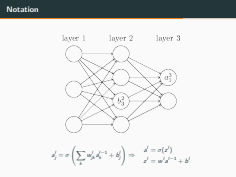
2020-04-11

Deep Learning

└ Aktuelle Entwicklung

└ Backpropagation

└ Notation



1. Ähnlich zu Gewichtsnotation
2. l bezieht sich hierbei jedoch auf aktuelle Schicht
3. j wie gehabt Index in Schicht
4. Notation gilt auch für Aktivierung a
5. Wichtig: σ bezieht sich auf Vektor \Rightarrow Vektorielle Funktion
6. Jede Komponente einzeln mit σ verarbeitet
7. Abstraktion vom Ausgabewert vor der Aktivierungsfkt. hilft später beim Ableiten

- Kostenfunktion soll minimiert werden
- Ziel: Optimale Gewichte und Schwellwerte finden
- Grobe Vorgehensweise: Iterativer Prozess
 - Fehlervektor der letzten Schicht berechnen
 - Fehler schichtweise zum Eingabelayer zurückführen
 - Parameter schichtweise nach Gradienten angleichen

2020-04-11

Deep Learning
└─ Aktuelle Entwicklung
 └─ Backpropagation
 └─ Backpropagation

Backpropagation

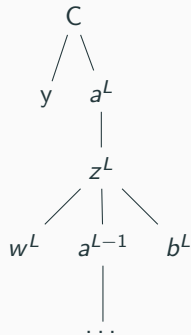
- Kostenfunktion soll minimiert werden
- Ziel: Optimale Gewichte und Schwellwerte finden
- Grobe Vorgehensweise: Iterativer Prozess
 - Fehlervektor der letzten Schicht berechnen
 - Fehler schichtweise zum Eingabelayer zurückführen
 - Parameter schichtweise nach Gradienten angleichen

1. Kostenfunktion wie bei Gradientenabstieg / Adeline
2. Unterschied: Hier mehrschichtiges Netz
3. 1970er entwickelt, 1986 von Rumelhart, Hinton und Williams in Paper bekannt gemacht
4. Gradientenabstieg grob erläutert, ausgeblieben - Anwendung im mehrschichtigen Netz und mehrdimensionale Kostenfunktion
5. Fehlervektor der letzten Schicht berechnen
6. Fehler schichtweise zum Eingabelayer zurückführen
7. Parameter schichtweise nach Gradienten angleichen

$$\begin{aligned}\delta_j^L &= \frac{\partial C}{\partial z_j^L} \\ &= \sum_k \frac{\partial C}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_j^L} \\ &= \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} \\ &= \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L)\end{aligned}$$

Anmerkung: Kettenregel

$$\frac{d}{dx} [f(u)] = \frac{d}{du} [f(u)] \frac{du}{dx}$$



- **C**: Kostenfunktion
- **y**: Erwartete Ausgabe

2020-04-11

Deep Learning

└ Aktuelle Entwicklung

└ Backpropagation

└ Fehler - Ausgabeschicht

$$\begin{aligned}\delta_j^L &= \frac{\partial C}{\partial z_j^L} \\ &= \sum_k \frac{\partial C}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_j^L} \\ &= \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} \\ &= \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L)\end{aligned}$$

Anmerkung: Kettenregel

$$\frac{d}{dx} [f(u)] = \frac{d}{du} [f(u)] \frac{du}{dx}$$



- **C**: Kostenfunktion
- **y**: Erwartete Ausgabe

1. Baum nur für Netz mit einer einzigen Aktivierung
2. Zusammenhang mit Kettenregel erläutern
3. Großes L immer für Ausgabeschicht

Fehler - Ausgabeschicht

2020-04-11

Deep Learning

└ Aktuelle Entwicklung

└└ Backpropagation

└└└ Fehler - Ausgabeschicht

Zusammenfassung

- Um den Fehlervektor der letzten Schicht zu bestimmen:

$$\delta^L = \nabla_a C \odot \sigma'(z^L)$$

- Äquivalent zu:

$$\delta^L = (a^L - y) \odot \sigma'(z^L)$$

- Um die Fehler komponentenweise zu bestimmen:

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

- $\nabla_a C$ entspricht dabei Vektor aller $\frac{\partial C}{\partial a_j^L}$ einer Schicht
- \odot : Komponentenweise Multiplikation zweier Vektoren

Zusammenfassung

- Um den Fehlervektor der letzten Schicht zu bestimmen:

$$\delta^L = \nabla_a C \odot \sigma'(z^L)$$

- Äquivalent zu:

$$\delta^L = (z^L - y) \odot \sigma'(z^L)$$

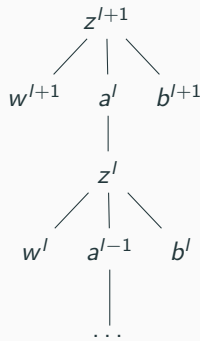
- Um die Fehler komponentenweise zu bestimmen:

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

Fehler - Zwischenschicht

- Zusammenhang zwischen Fehler zweier Schichten herleiten
- Es gilt: $\delta_j^l = \partial C / \partial z_j^l$ sowie $\delta_k^{l+1} = \partial C / \partial z_k^{l+1}$

$$\begin{aligned}\delta_j^l &= \frac{\partial C}{\partial z_j^l} \\ &= \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} \\ &= \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1}\end{aligned}$$



2020-04-11

Deep Learning

└ Aktuelle Entwicklung

└ Backpropagation

└ Fehler - Zwischenschicht

• Zusammenhang zwischen Fehler zweier Schichten herleiten

• Es gilt: $\delta_j^l = \partial C / \partial z_j^l$ sowie $\delta_k^{l+1} = \partial C / \partial z_k^{l+1}$



$$z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1} \sigma(z_j^l) + b_k^{l+1}$$

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l)$$

Zusammenfassung

- Komponentenweise Darstellung: $\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l)$
- Vektorielle Darstellung: $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$

2020-04-11

Deep Learning

└ Aktuelle Entwicklung

└ Backpropagation

└ Fehler - Zwischenschicht

Fehler - Zwischenschicht

$$z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1} \sigma(z_j^l) + b_k^{l+1}$$

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l)$$

Zusammenfassung

- Komponentenweise Darstellung: $\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l)$

- Vektorielle Darstellung: $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$

Fehler - Schwellwerte & Gewichte

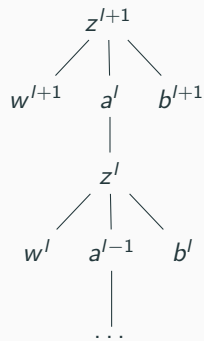
$$z_k^l = \sum_j w_{kj}^l a_j^{l-1} + b_k^l = \sum_j w_{kj}^l \sigma(z_j^{l-1}) + b_k^l$$

Schwellwerte

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l$$

Gewichte

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l$$



2020-04-11

Deep Learning

└ Aktuelle Entwicklung

└ Backpropagation

└ Fehler - Schwellwerte & Gewichte

Fehler - Schwellwerte & Gewichte

$$z_k^l = \sum_j w_{kj}^l a_j^{l-1} + b_k^l = \sum_j w_{kj}^l \sigma(z_j^{l-1}) + b_k^l$$

Schwellwerte

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l$$

Gewichte

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l$$



- Menge an Trainingsdatensätzen auswählen
- Für jeden einzelnen Datensatz:
 1. **Feedforward:** Z-Wert und Aktivierung für jede Schicht $l = 2, 3, \dots, L$ berechnen.
 - Z-Wert: $z^{x,l} = w^l a^{l-1} + b^l$
 - Aktivierung $a^{x,l} = \sigma(z^l)$
 2. **Ausgabe-Fehler** $\delta^{x,L}$: Fehlervektor der Ausgabeschicht berechnen.
 - $\delta^L = \nabla_a C \odot \sigma'(z^L)$
 3. **Backpropagation-Fehler:** Rückwirkend Fehlervektor aller Schichten berechnen.
 - $\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l})$
- **Gradientenabstieg:** Gewichte und Schwellwerte getrennt anpassen.
 - Gewichte: $w^l \rightarrow w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$
 - Schwellwerte: $b^l \rightarrow b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$

2020-04-11

Deep Learning
└ Aktuelle Entwicklung
└ Backpropagation
└ Anwendung

Anwendung

- Menge an Trainingsdatensätzen auswählen
- Für jeden einzelnen Datensatz:
 1. **Feedforward:** Z-Wert und Aktivierung für jede Schicht $l = 2, 3, \dots, L$ berechnen.
 - Z-Wert: $z^l = w^l a^{l-1} + b^l$
 - Aktivierung $a^l = \sigma(z^l)$
 2. **Ausgabe-Fehler** $\delta^{x,L}$: Fehlervektor der Ausgabeschicht berechnen.
 - $\delta^L = \nabla_a C \odot \sigma'(z^L)$
 3. **Backpropagation-Fehler:** Rückwirkend Fehlervektor aller Schichten berechnen.
 - $\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l})$
- **Gradientenabstieg:** Gewichte und Schwellwerte getrennt anpassen.
 - Gewichte: $w^l \rightarrow w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$
 - Schwellwerte: $b^l \rightarrow b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$

Aktuelle Entwicklung

Multilayer Perceptron

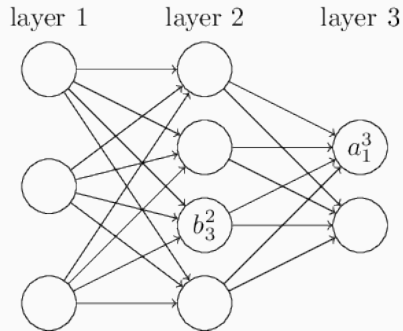
2020-04-11

Deep Learning
└─ Aktuelle Entwicklung
 └─ Multilayer Perceptron

Aktuelle Entwicklung
Multilayer Perceptron

Multilayer Perceptron

- In Grundzügen bereits beim Backpropagation Algorithmus erläutert



Anwendungsbereiche:

- Mustererkennung
- Funktionenapproximation
- Klassifizierung
- Prognose
- Diagnose
- Steuerung
- Optimierung

2020-04-11

Deep Learning

└ Aktuelle Entwicklung

└ Multilayer Perceptron

└ Multilayer Perceptron

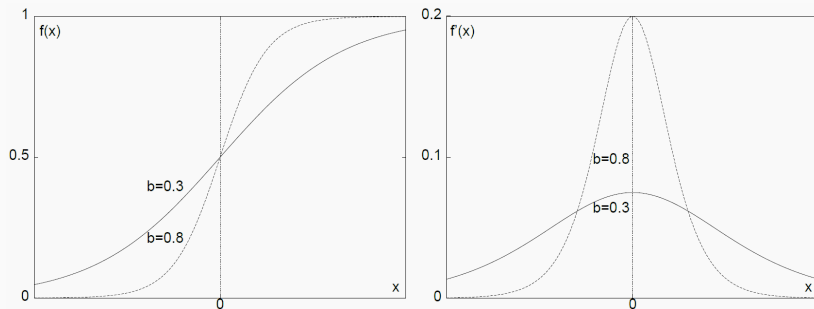


1. vielfältige Struktur
2. Mehrschichtiges Netz, bereits beim BPAlgo. verwendet
3. Zu tiefe Netz: Probleme beim Training
4. Techniken unter Deep learning zusammengefasst
5. Stuktur: mehrschichtiges vorwärtsgekoppeltes Netz
6. Feedforward: durchiterieren von Eingabewerten
- 7.
- 8.

Sigmoid Aktivierungsfunktion

- Einfach / schnell zu berechnen
- Einfach / schnell abzuleiten

$$f(x) = \frac{1}{1 + \exp(-b * x)}$$
$$f'(x) = b * f(x)(1 - f(x))$$



2020-04-11

Deep Learning

└ Aktuelle Entwicklung

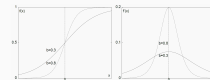
└ Multilayer Perceptron

└ Sigmoid Aktivierungsfunktion

Sigmoid Aktivierungsfunktion

- Einfach / schnell zu berechnen
- Einfach / schnell abzuleiten

$$f(x) = \frac{1}{1 + \exp(-b * x)}$$
$$f'(x) = b * f(x)(1 - f(x))$$



1. Wertebereich: zwischen 0 und 1
2. Konstante beschreibt Steilheit der Kurve
3. Über die komplette Domäne differenzierbar

2020-04-11

Deep Learning

└ Aktuelle Entwicklung

└ Recurrent Neural Network

Aktuelle Entwicklung

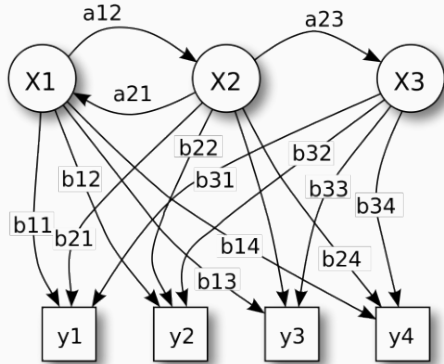
Recurrent Neural Network

Aktuelle Entwicklung

Recurrent Neural Network

Recurrent Neural Network

- Bekommen sowohl von Eingangssignale sowohl durch Trainingsdatensatz als auch durch Rückkopplungen.



- Mustererkennung
- Muster ergänzen
- Sprachanalyse

2020-04-11

Deep Learning

- └ Aktuelle Entwicklung
 - └ Recurrent Neural Network
 - └ Recurrent Neural Network

• Bekommen sowohl von Eingangssignale sowohl durch Trainingsdatensatz als auch durch Rückkopplungen.



- Mustererkennung
- Muster ergänzen
- Sprachanalyse

1. Bisher nur feedforward NN
2. Hierbei Reihenfolge von Bedeutung



Deep Learning

Einführung - Thema 2

Silas Hoffmann

11. April 2020

Fachhochschule Wedel

2020-04-11

Deep Learning

└ Aktuelle Entwicklung

└ Recurrent Neural Network



Deep Learning

Einführung - Thema 2

Silas Hoffmann
11. April 2020
Fachhochschule Wedel

2020-04-11

Deep Learning

└ Aktuelle Entwicklung

└ Recurrent Neural Network

Alle Materialien sind unter folgender URL zu finden:
https://github.com/derMacon/deeplearning_seminar

Alle Materialien sind unter folgender URL zu finden:

`https://github.com/derMacon/deeplearning_seminar`

Sometimes, it is useful to add slides at the end of your presentation to refer to during audience questions.

The best way to do this is to include the `appendixnumberbeamer` package in your preamble and call `\appendix` before your backup slides.

metropolis will automatically turn off slide numbering and progress bars for slides in the appendix.

Backup slides

Sometimes, it is useful to add slides at the end of your presentation to refer to during audience questions.

The best way to do this is to include the `appendixnumberbeamer` package in your preamble and call `\appendix` before your backup slides.

metropolis will automatically turn off slide numbering and progress bars for slides in the appendix.



3Blue1Brown - Videokurs zur Einführung in die Neuralen Netze.

https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi.

Aufgerufen am: 16-03-2020.



Übersicht - verschiedene Architekturen.

<https://www.asimovinstitute.org/neural-network-zoo/>.

Aufgerufen am: 22-03-2020.



Definition Klassifizierungsproblem.

http://ekpwww.physik.uni-karlsruhe.de/~tkuhr/HauptseminarWS1112/Keck_handout.pdf.

Aufgerufen am: 15-03-2020.



Einführung Convolutional neural network.

<https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/>.

Aufgerufen am: 18-03-2020.



Öffentliche Datensätze - Übersicht.

<https://github.com/awesomedata/awesome-public-datasets>.

Aufgerufen am: 18-03-2020.



Funktionsweise - CNN.

<https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1890437/>.

Aufgerufen am: 18-03-2020.



Funktionsweise - CNN.

<https://bit.ly/2QGK0Ej>.

Aufgerufen am: 18-03-2020.



References



Geschichte der Convolutional neuronalen Netze.

<https://glassboxmedicine.com/2019/04/13/a-short-history-of-convolutional-neural-networks/>.

Aufgerufen am: 18-03-2020.



Khan Academy - Partielle Ableitungen (Funktion mit zwei Eingabewerten.

<https://www.youtube.com/watch?v=1CMDS4-PKKQ&t=542s>.

Aufgerufen am: 16-03-2020.



Künstliche Neuronale Netzwerke und Deep Learning - Stefan Stelle.

https://www.htwsaar.de/wiwi/fakultaet/personen/profile/selle-stefan/Selle2018e_Kuenstliche_Neuronale_Netzwerke.pdf/at_download/file.

Aufgerufen am: 24-03-2020.



McCulloch-Pitts Neuron.

<https://towardsdatascience.com/mcculloch-pitts-model-5fdf65ac5dd1>.

Aufgerufen am: 14-03-2020.



Perceptron - Python Implementierung.

<https://github.com/rasbt/mlxtend/blob/master/mlxtend/classifier/perceptron.py>.

Aufgerufen am: 16-03-2020.



Single-Layer Neural Networks and Gradient Descent.

https://sebastianraschka.com/Articles/2015_singlelayer_neurons.html.




Aufgerufen am: 14-03-2020.



M. Nielsen.

Neural Networks and Deep Learning.

Determination Press, 2015.

-  McCulloch-Pitts Neuron.
<https://towardsdatascience.com/mcculloch-pitts-model-5fdf65ac5dd1>.
Aufgerufen am: 14-03-2020.
-  Perceptron - Python Implementierung.
<https://github.com/rasbt/mlxtend/blob/master/mlxtend/classifier/perceptron.py>.
Aufgerufen am: 16-03-2020.
-  Single-Layer Neural Networks and Gradient Descent.
https://sebastianraschka.com/Articles/2015_singlelayer_neurons.html.
Aufgerufen am: 14-03-2020.
-  M. Nielsen.
Neural Networks and Deep Learning.
Determination Press, 2015.