Deep Learning

Einführung - Thema 2

Silas Hoffmann

13. April 2020

Fachhochschule Wedel

Deep Learning

2020-04-13

Deep Learning

Einführung - Thema 2

Silas Hoffmann

13. April 2020

Fachhochschule Wedel

Inhalt

Inhalt

Geschichtliche Entwicklung
McCulloch-Pitts-Neuron
Perceptron
Adeline

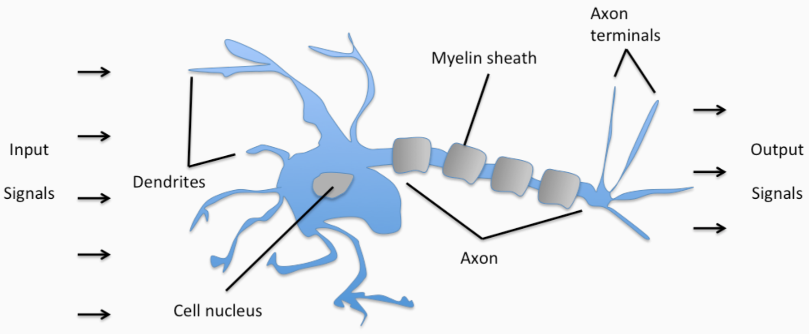
Aktuelle Entwicklung
Convolutional Neural Network

Geschichtliche Entwicklung

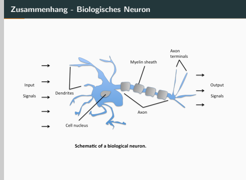
McCulloch-Pitts-Neuron

Geschichtliche Entwicklung
McCulloch-Pitts-Neuron

Zusammenhang - Biologisches Neuron



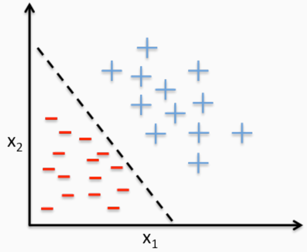
Schematic of a biological neuron.



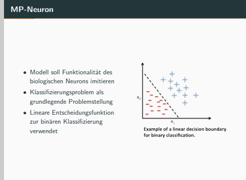
1. *Dendriten*: Nehmen Infos auf
   - besizten Rezeptoren und Signale anderer Dendriten aufzunehmen
2. Signale: bewirken elektrische Veränderungen
   - werden vom Zellkern (Soma) interpretiert / verarbeitet
   - Zellkern sammelt Infos, speichert diese im Axonhügel
3. Ursprung vom Axon / Neuriten
4. Wenn Signal stark genug: an Axon weitergeleitet
   - auch als *Aktionspotential bezeichnet*
   - Signal am Ende über Axonterminale per Neurotransmitter mit nächste Dendriten verbunden

MP-Neuron

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

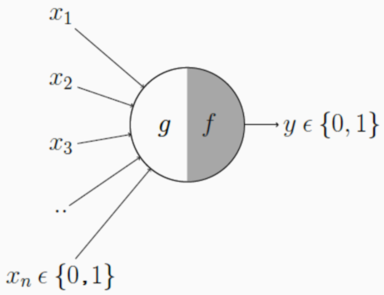


Example of a linear decision boundary for binary classification.

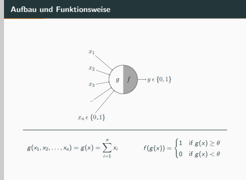


1. 1943: Warren McCulloch & Walter Pitts
2. soll biologisches Neuron imitieren
3. Klassifizierungsproblem: anhand vom geg. Merkmalsvektor entscheiden ob Objekt X in Klasse K liegt
4. hier lediglich binäre Klassifikation
   - Unterscheidung nur zwischen zwei Klassen
   - Sonderfall dieses Modells: nur boolesche Eingabewerte
5. muss mittels linearer Entscheidungsfunktion definierbar sein

Aufbau und Funktionsweise

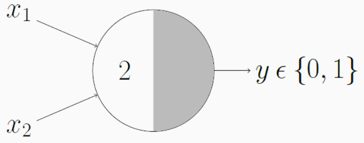


$$g(x_1, x_2, \ldots, x_n) = g(x) = \sum_{i=1}^{n} x_i \qquad f(g(x)) = \begin{cases} 1 & \text{if } g(x) \geq \theta \\ 0 & \text{if } g(x) < \theta \end{cases}$$



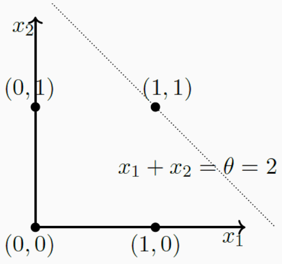
1. beliebig viele Eigabewerte
   – müssen boolescher Natur sein
2. Arbeitsschritte:
   – Alle Werte aufaddiert (Fkt. g)
   – Fkt. f prüft ob Schwellwert überschritten

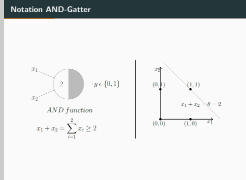
Notation AND-Gatter



$AND\ function$

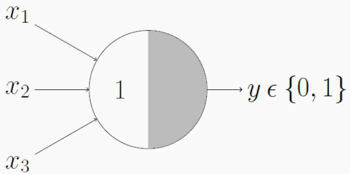
$$x_1 + x_2 = \sum_{i=1}^{2} x_i \geq 2$$





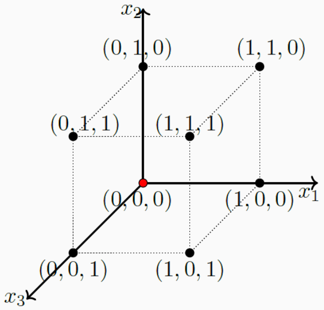
1. Anhand von Grafik erläutern

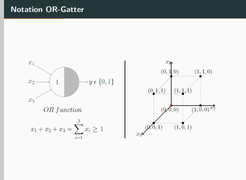
Notation OR-Gatter



OR function

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

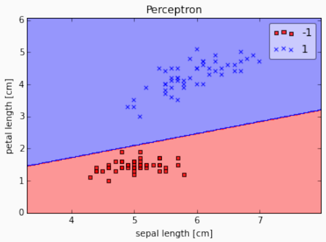


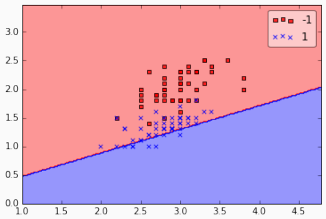


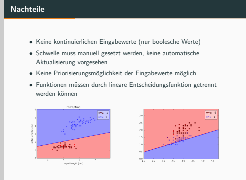
1. Anhand von Grafik erläutern, auch im 3d - Raum möglich

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







1. keine kontinuierlichen Eignabewerte
   - nur boolesche Werte
   - Schwierig für komplexe Anwendungen
   - siehe Bilderkennung - Farbwerte
2. Schwelle muss manuelle gesetzt werden
   - Sprich kein Lernalgorithmus vorhanden
3. Keine Priorisierungsmöglichkeiten
   - siehe Gewichtete Eingaben
4. Funktionen durch lineare Entscheidungsfunktion getrennt
   - schwierig bei überlappenden Cluster
   - keine Polynome wie bei späteren Entwicklungen möglich
5. auch gedeckelte Fkt. wie XOR können nicht dargestellt werden
   - Schwelle muss genau getroffen werden

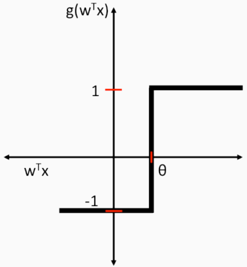
Geschichtliche Entwicklung

Perceptron

Geschichtliche Entwicklung
Perceptron

Perceptron

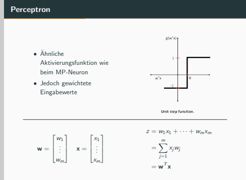
- Ähnliche Aktivierungsfunktion wie beim MP-Neuron
- Jedoch gewichtete Eingabewerte



Unit step function.

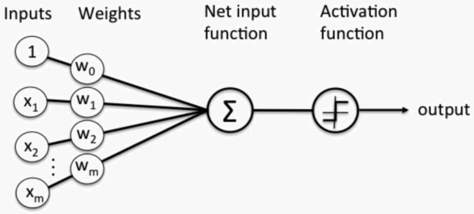
$$\mathbf{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$$

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



1. 1958: US-amerikanische Psychologe / Informatiker Frank Rosenblatt
2. älteste heutzutage noch genutzte NN
3. inspiriert vom Auge einer Fliege
   - Flugrichtung - Entscheidungen teils direkt im Auge getroffen
4. Weiterentwicklung der MP-Zelle
5. Eingabewerte mit Gewichten priorisiert
   - Auf Formel verweisen
6. Gleich bleibt jedoch die binäre Klassifikation
   - Verweis auf Unit step function
   - hier jedoch nicht Wahrheitswerte sondern -1 und 1

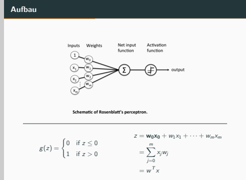
Aufbau



Schematic of Rosenblatt's perceptron.

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

$$\begin{aligned} z &= \mathbf{w_0 x_0} + w_1 x_1 + \dots + w_m x_m \\ &= \sum_{j=0}^{m} x_j w_j \\ &= w^T x \end{aligned}$$



1. Grafik erläutern
2. Konvention:
   - erleichtert später Notation der Lernregel
   - Schwellwert auf andere Seite der z-Wert Gleichung ziehen

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

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

1. Rosenblatt erfindet *lernenden* Algorithmus
2. Auf Menge von Trainingsdatensätzen zurückgegriffen
   - Datensätze bestehen aus Ein- und erwarteten Ausgabewerten
   - in Literatur auch *gelabelte* Werte genannt
3. Lernalgorithmus - grobe Zusammenfassung
   - Gewichte mit kleinem Wert / 0 vorinitialisieren
   - Datensätze durchiterieren
     - Ausgabewert berechnen
     - Gewichte angleichen

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:

$$\Delta w_0 = \eta(\text{target}^{(i)} - \text{output}^{(i)})$$
$$\Delta w_1 = \eta(\text{target}^{(i)} - \text{output}^{(i)})\ x_1^{(i)}$$
$$\Delta w_2 = \eta(\text{target}^{(i)} - \text{output}^{(i)})\ x_2^{(i)}$$

1. Erste Formel auf Slide beschreiben
   - Gewichte können zu Gewichtsvektor zusammengezogen werden
     - hier komponentenweise betrachtet
   - Delta (Dreieck) wird stets als Änderung verstanden
2. Exponent *i* hierbei jeweils als Index des Trainingsvektors in Menge
3. Lernalgorithmus arbeitet inkrementell
   - Lernrate (eta) bestimmt wie stark die Gewichte pro Durchlauf angeglichen werden
   - Differenz mit Lernrate und Eingabewert multipliziert
4. Iteration mit 2d - Eingabevektor
   - w0 hierbei der Schwellwert selbst
   - Faktor x weggelassen da bereits gleich 1
   - Nutzung der beschriebenen Notation

Lernregel - Trainingsbeispiele

Gewichtsänderung

$\Delta w_j = \eta \, (\text{target}^{(i)} - \text{output}^{(i)}) \, x_j^{(i)}$

- Trainingsdatensatz richtig erkannt:

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

- Trainingsdatensatz falsch erkannt:

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

Lernregel - Trainingsbeispiele
Gewichtsänderung
$\Delta w_j = \eta \, (\text{target}^{(i)} - \text{output}^{(i)}) \, x_j^{(i)}$
Trainingsdatensatz richtig erkannt:
$\Delta w_j = \eta\big((-1^{(i)}) - (-1^{(i)})\big) \, x_j^{(i)} = 0$
$\Delta w_j = \eta(1^{(i)} - 1^{(i)}) \, x_j^{(i)} = 0$
Trainingsdatensatz falsch erkannt:
$\Delta w_j = \eta\big(1^{(i)} - (-1^{(i)})\big) \, x_j^{(i)} = \eta(2) \, x_j^{(i)}$
$\Delta w_j = \eta\big((-1^{(i)}) - 1^{(i)}\big) \, x_j^{(i)} = \eta(-2) \, x_j^{(i)}$

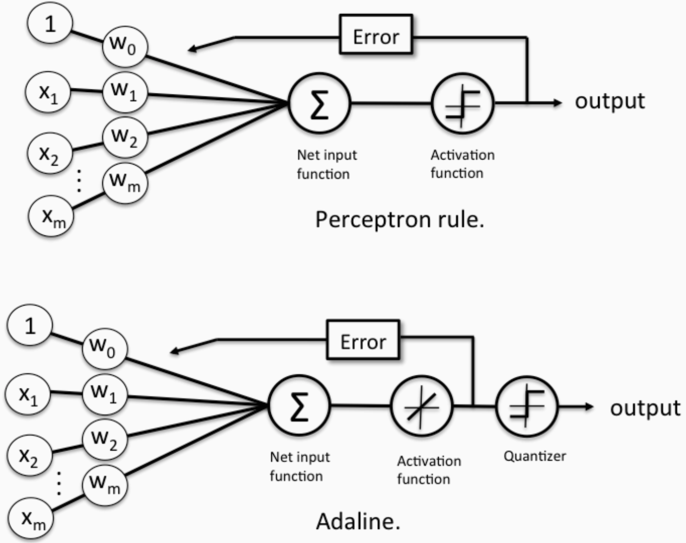
1. Erinnerung: erst target dann output
2. Richtig erkannt
   - Generell Ausgabe 0, keine Änderung
   - Beide Falsch: -1
   - Beide Richtig: +1
3. Falsch erkannt
   - output zu klein
     - erwartetet +1 bekommen -1
     - Positiver (Differenz-)Faktor
   - output zu groß
     - erwartetet -1 bekommen +1
     - Negativer (Differen-)Faktor

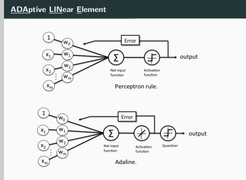
Geschichtliche Entwicklung

Adeline

Geschichtliche Entwicklung
Adeline

ADAptive LINear Element





1. 1959: Stanford Prof. Bernard Widrow & Elektroingenieur Marcian Edward Hoff
2. ADELINE: ADAptive LINear Element
3. Modell: Verzicht auf Einheitssprungfunktion bei Angleichung der Gewichte

   – Stattdessen lineare Aktierungsfunktion
   – erstmal nur Identitätsfunktion verwendet
   – Entscheidungsfunktion für output weiterhin verwendet

Delta-Regel

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

Notation

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

Delta-Regel

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

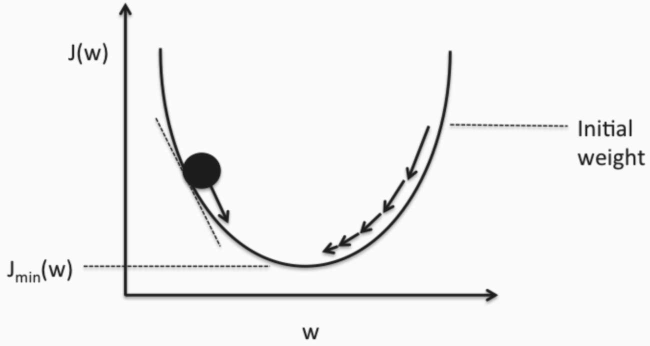
Notation

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

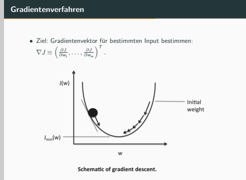
1. Auch unter *Least-Mean-Square-Algorithmus* bzw. *Regressionsquadratsumme* bekannt
   - noch heute relevant
2. Funktion stellt Kostenfunktion dar
   - Fehler bei Kostenfunktion soll mithilfe der Lernregel minimiert werden
3. Vorteil dieses Ansatzes: Ableitbare Kostenfunktion
4. Formel erläutern:
   - Differenz quadriert um Vorzeichen zu verlieren
   - Faktor 1 / 2 vorschieben um Ableitung einfacher zu gestalten
   - über alle Trainingsdatensätze der Menge iterieren
     - Größe i
5. Für genaueres Verständnis erstmal Einschub mit 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.



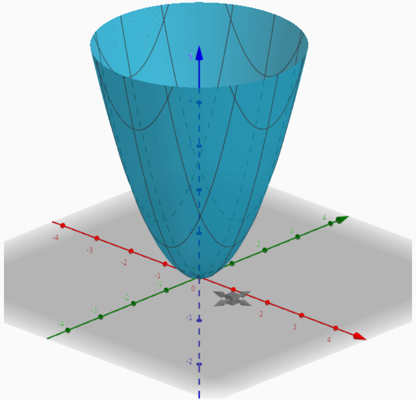
1. Wesentlicher Nachteil der Sprungfunktion: Nicht stetig & damit nicht differenzierbar
2. Adeline verwendet Identitätsfunktion
3. Abbildung erläutern, Metapher: Ball rollt Hügel herunter
   - Abbildung erstmal nur mit einem einzelnen Gewicht geplottet
   - Ableitung an einer bestimmten Stelle gleich der Steigung
   - Gradientenvektor gibt diese Richtung an
     - Mehrdimensional wenn mehreren Eingabeargumenten vorhanden
   - Steigung muss invertiert werden
4. Es folgt: Exkurs - Partielle Ableitungen

Partielle Ableitungen

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

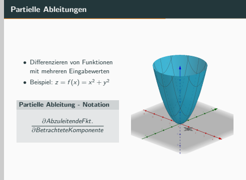
Partielle Ableitung - Notation

$$\frac{\partial AbzuleitendeFkt.}{\partial BetrachteteKomponente}$$

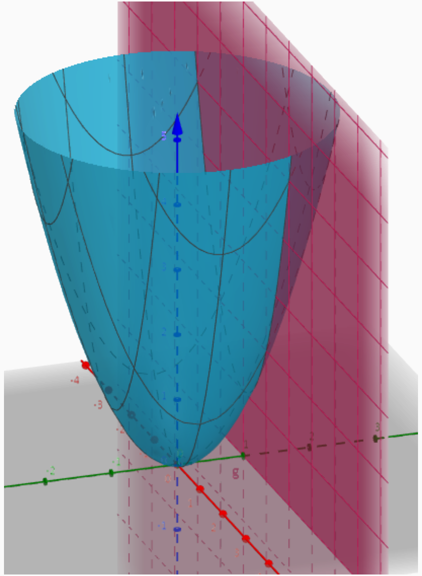


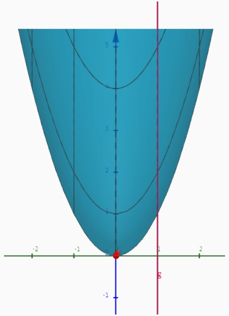
2020-04-13

Deep Learning
└Geschichtliche Entwicklung
  └Adeline
    └Partielle Ableitungen



1. Notation: Bruch
   - Zähler: Abzuleitende Funktion
   - Nenner: Betrachtete Komponente
2. Abbildung: Fkt. geplottet mit 2 Eingabekomponenten
   - Funktion: $z = f(x) = x^2 + y^2$
   - Metapher: Blickwinkel erläutern
   - nächste Folie miteinbeziehen

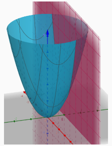


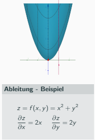


Ableitung - Beispiel

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

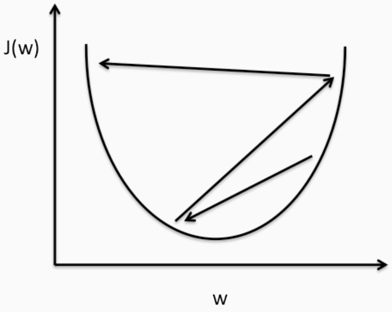
$$\frac{\partial z}{\partial x} = 2x \qquad \frac{\partial z}{\partial y} = 2y$$



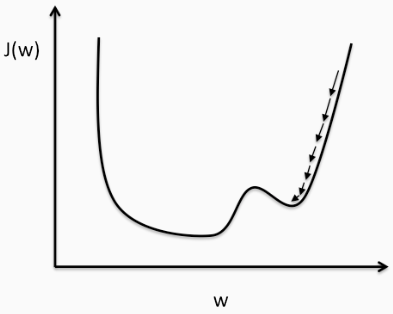


1. Metapher: Blickwinkel erläutern

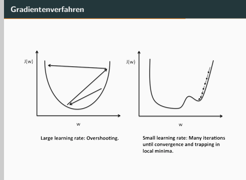
Gradientenverfahren



Large learning rate: Overshooting.

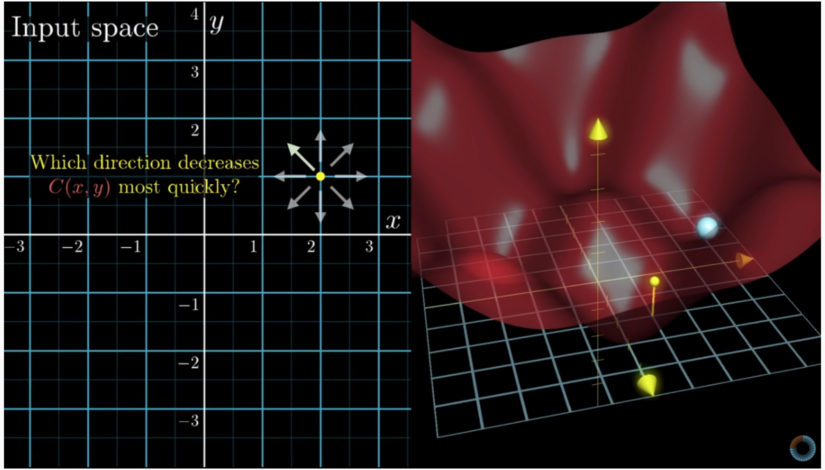


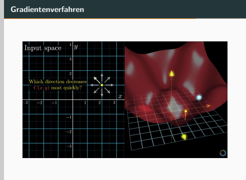
Small learning rate: Many iterations until convergence and trapping in local minima.



1. Lernrate kann als Schrittweite verstanden werden
2. Zwei mögliche Probleme:
   - Overshooting: Schrittweite zu groß - Minimum wird nicht erkannt
   - Lokales Minimum wird gefunden - Globales bleibt unerkannt
3. Gradientenabstieg bisher nur in 2 Dimensionen (siehe nächste Folie

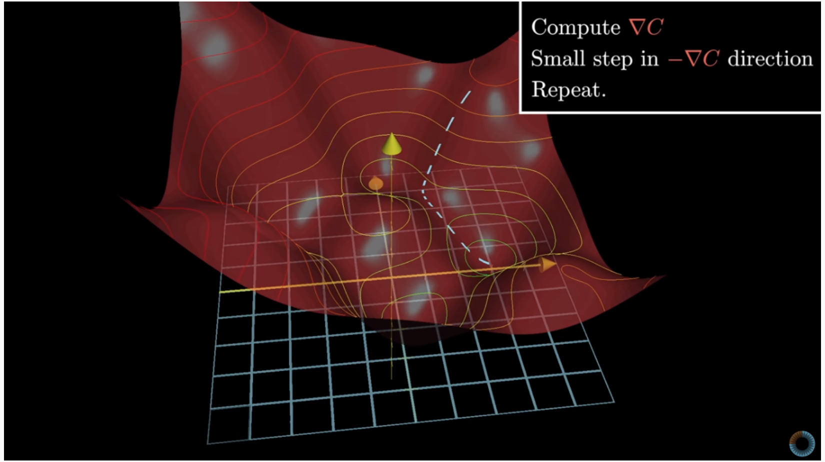
Gradientenverfahren

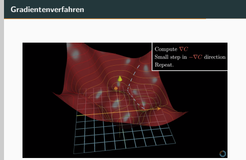




1. Abbildung: Gradientenabstieg in 3 Dimensionen geplottet
2. Hier Ball-Metapher dargestellt
3. Es folgt kompletter Durchlauf des Gradientenabsiegs

Gradientenverfahren





1. Abbildung: Gradientenabstieg in 3 Dimensionen geplottet
2. Hier durchgeführter Gradientenabstieg

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 $w = w + \Delta w$

1. Gradientenvektor: Richtung des *Abstiegs*
   - mit *Nabla* dargestellt (Dreieck)
   - kann auch mehrdimensional sein
2. Vektorielle Darstellung
   - Eingabeparameter werden als Vektor verstanden
   - mit Gradientenvektor und Negativer Lernrate verrechnet / multipliziert
3. Komponentenweise Darstellung
   - negative Lernrate mit partieller Ableitung verrechnet
4. Angleichung der Gewichte:
   - wie schon bei vorherigen Modellen
   - Mathematische Darstellung: $w = w + \Delta w$

Kostenfunktion ableiten

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

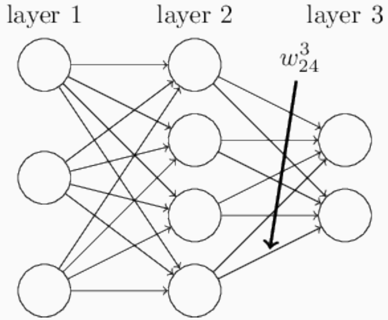
1. Ableiten der bisher vorgestellten Kostenfuntion (Least-Mean-Square)
2. Summe und Faktor vorziehen
3. Kettenregel anwenden
   - äußere Ableitung bereits bestimmt (Vorfaktor 2)
   - innere Ableitung steht noch aus
4. Faktor 2 kann vorgezogen werden, wird mit 1/2 verrechnet
5. Ursprüngliche Notation für die Ausgabe wird eingesetzt:
   - Ausgabe: $\sum_j w_j x_j^{(i)}$
6. Summe aufgelöst
   - es wird nach w_j abgeleitet
   - alle Summanden in denen dieser Faktor nicht vorkommt entfallen

Aktuelle Entwicklung

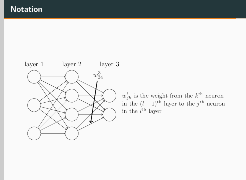
Backpropagation

Aktuelle Entwicklung
Backpropagation

Notation

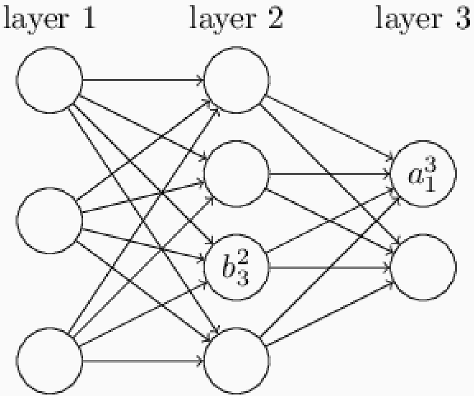


w_{jk}^{l} is the weight from the k^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer

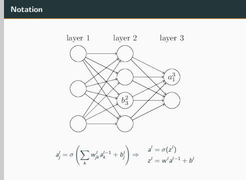


1. l: Exponent, steht für die Schicht
   - l - 1, weil man stets von hinten nach vorne schaut
2. Eingabe wird auch als eigene Schicht verstanden
3. j: Index Zielneuron
4. k: Index Startneuron

Notation



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



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

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

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

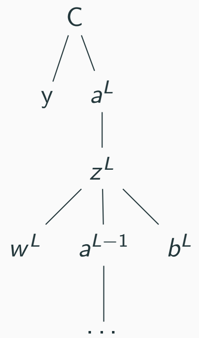
1. 1970er entwickelt, 1986 von Rummelhart, Hilten und Williams in Paper bekannt gemacht
2. Kostenfunktion wie bei Gradientenabstieg / Adeline
   - Unterschied: Hier mehrschichtiges Netz
   - Gradientenabstieg grob erläutert, ausgeblieben -
     - Anwendung im mehrschichtigen Netz und mehrdimensionale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

Fehler - Ausgabeschicht

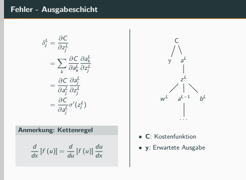
$$\begin{aligned}
\delta_j^L &= \frac{\partial C}{\partial z_j^L} \\
&= \sum_k \frac{\partial C}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_j^L} \\
&= \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} \\
&= \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L)
\end{aligned}$$

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 repräsentativ
2. Zusammenhang mit Kettenregel erläutern
3. Großes L immer für Ausgabeschicht
4. Summfunktion: für mehrere Neuronen pro Schicht generalisiert

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:

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

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

1. Um den Fehlervektor der letzten Schicht zu bestimmen: $\delta^L = \nabla_a C \odot \sigma'(z^L)$
   - $\nabla_a C$ entspricht dabei Vektor aller $\frac{\partial C}{\partial a_j^L}$ einer Schicht
   - $\odot$: Hadamard-Produkt
     - Komponentenweise Multiplikation zweier Vektoren
     - Ausgabe ebenfalls wieder ein Vektor
2. Äquivalent zu: $\delta^L = (a^L - y) \odot \sigma'(z^L)$
   - $(a^L - y)$ Ausgabe des Systems minus erwartete Ausgabe
3. Um die Fehler komponentenweise zu bestimmen: $\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L)$

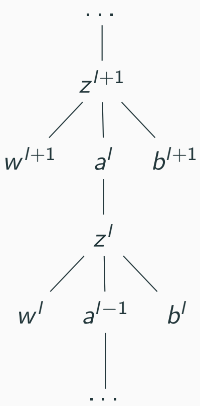
Fehler - Zwischenschicht

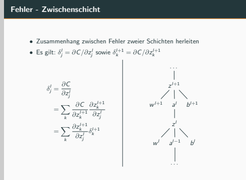
- Zusammenhang zwischen Fehler zweier Schichten herleiten
- Es gilt: $\delta_j^l = \partial C / \partial z_j^l$ sowie $\delta_k^{l+1} = \partial C / \partial z_k^{l+1}$

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

$$= \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l}$$

$$= \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1}$$





1. Um von letzter Schicht auf vorherige Fehler zu schließen: $\delta_k^{l+1} = \partial C / \partial z_k^{l+1}$
2. Baum: Netz-Ausschnitt mit nur einem Neuron pro Schicht
3. Über Kettenregel wird nach aktuellem Z-Wert abgeleitet
4. Reihenfolge vertauscht
   - Letzter Term mit Definition $\delta_k^{l+1} = \partial C / \partial z_k^{l+1}$ ausgetauscht

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

1. Zwischenschritt: Z-Wert des nächsten Layers
   - Definition Z-Wert eingesetzt
   - Aktivierung wird mit Sigma-Funktion ausgetauscht
2. Diese Gleichung wird nun nach ∂z_j^l abgeleitet
3. Komponentenweise Darstellung: $\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l)$
   - vorherigen Zwischenschritte wurden wieder in die ursprüngliche Form eingesetzt $\sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1}$
   - Reihenfolge der Faktoren wurde lediglich etwas verändert
4. Vektorielle Darstellung: $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$
   - Summfunktion durch Vektormultiplikation ausgetauscht

Fehler - Schwellwerte & Gewichte

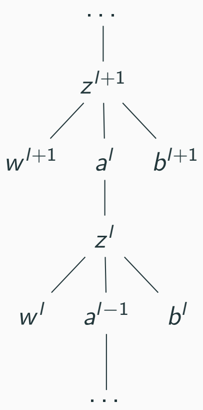
$$z^l_k = \sum_j w^l_{kj} a^{l-1}_j + b^l_k = \sum_j w^l_{kj} \sigma(z^{l-1}_j) + b^l_k$$

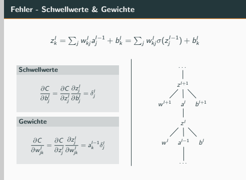
Schwellwerte

$$\frac{\partial C}{\partial b^l_j} = \frac{\partial C}{\partial z^l_j} \frac{\partial z^l_j}{\partial b^l_j} = \delta^l_j$$

Gewichte

$$\frac{\partial C}{\partial w^l_{jk}} = \frac{\partial C}{\partial z^l_j} \frac{\partial z^l_j}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$$





1. Bisher nur die Ableitung nach Z-Wert betrachtet
   – Nun nach Schwellwerten & Gewichten
2. Kettenregel: Ableitung nach dem Z-Wert vor Ableitung nach Gewicht / Schwellwert schalten
3. **Schwellwerte**: der hintere Bruch entfällt komplett
   – vordere Teil $\frac{\partial C}{\partial z^l_j}$ entspricht bereits δ^l_j
4. **Gewichte**: obere Z-Wert Gleichung nach Gewicht abgeleitet
   – $w^l_{kj} a^{l-1}_j \rightarrow a^{l-1}_k$
   – Fehler per Kettenregel angehängt

Anwendung

- Menge an Trainingsdatensätzen auswählen
- Für jeden einzelnen Datensatz:
  1. **Feedforward**: Z-Wert und Aktivierung für jede Schicht $l = 2, 3, \ldots, L$ 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: $b^l \rightarrow b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$

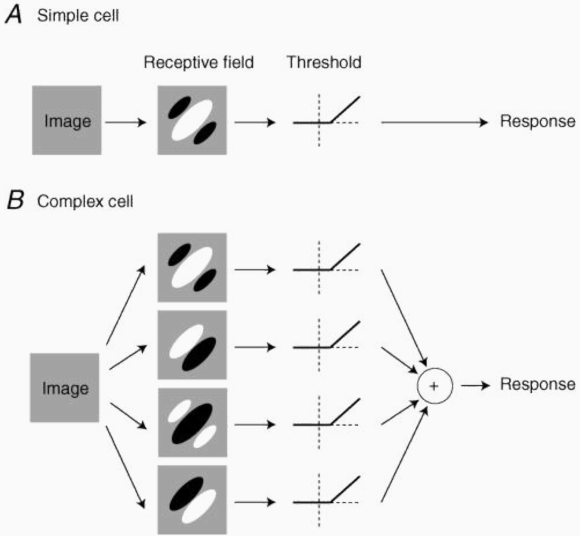
1. Menge an Trainingsdatensätzen auswählen
   – Siehe Stachastischer Gradientenabstieg
2. Für jeden Datensatz
   – *Feedforward* Aktivierungsvektor / Z-Wert jeder Schicht berechnen
   – *Ausgabe-Fehler*: Fehler letzter Schicht berechnen
   – *Backpropagation-Fehler*: Ausgehend von letzter Schicht Fehler bis hin zur Ersten berechnen
3. Gradientenabstieg mit Ergebnissen
   – Gewichte: $w^l \rightarrow w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$
     - hintere Teil: durchschnittlicher Fehler über alle m Trainingsdatensätze
     - Lernrate als Faktor davorgehängt
   – Schwellwerte: $b^l \rightarrow b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$

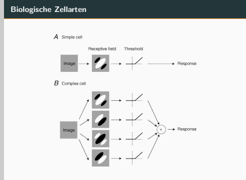
Aktuelle Entwicklung

Convolutional Neural Network

Aktuelle Entwicklung
Convolutional Neural Network

Biologische Zellarten

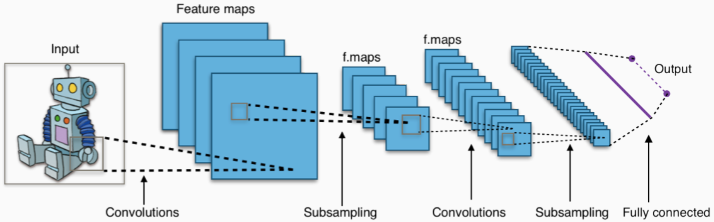


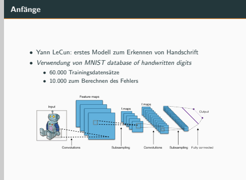


1. 1962: zwei Neurophysiologen Torsten Wiesel und David Hubel
2. Konzept der simple und complex cells
   – nicht positionsbunden - spatial invariance, räumliche Invarianz
3. Arten von Zellen zur Erkennung einfacher Kanten und Balken
   – *simple cells*: ist Positionsgebunden
   – *complex cells*: Muster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

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

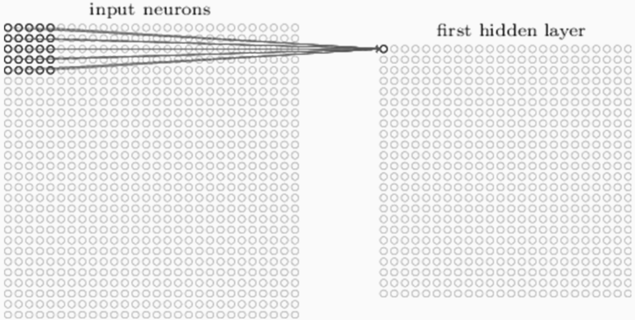




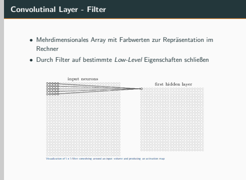
1. Pioniere, fr. Informatiker Yann LeCun
2. Bekannteste Ausarbeitung über CNN für Handschriften
3. *Verwendung von MNIST database of handwritten digits*
   - 60.000 Trainingsdatensätze
   - 10.000 zum Berechnen des Fehlers
   - unterschiedliche Personen für Trainings- und Evaluierungsdatensätze
4. soll erkennen ob ein Bild zu einer (oder mehreren) bestimmten Klasse(n) gehört

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



1. Array als Eingabe
   - Repräsentiert die Pixel im Bild
2. Farbwertearray kann pro Pixel mehrere Werte enthalten
   - entsprechend eventuell auch mehrere Dimensionen im Array

Filter

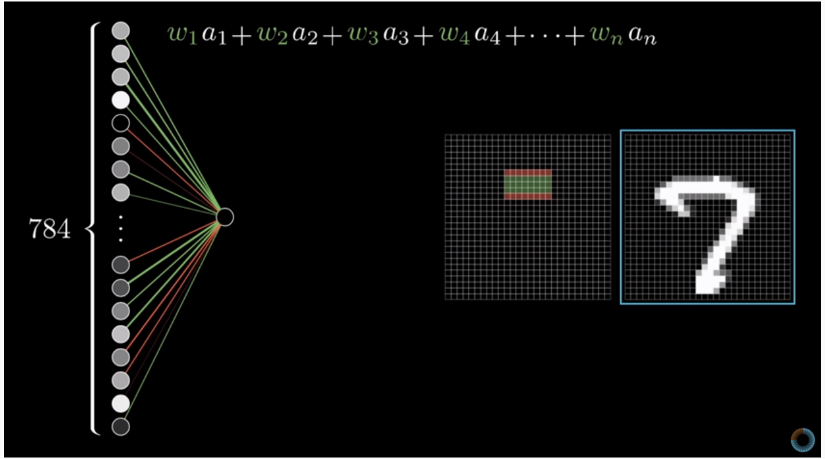
- Generell
  - Besitzt feste Pixelgröße (*Kernelsize*) & Schrittweite
  - Scannt Bild Zeilenweise
  - *Padding* legt Verfahren für Rand des Bildes fest
  - Ausgabe wird *activation* oder *feature map* genannt
- Praxis
  - *Convolutional Layer* mit 32 oder 16 Bit
  - Jeder Filter generiert eigene Ausgabematrix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

Filter

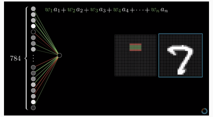
- Generell
  - Besitzt feste Pixelgröße (*Kernelsize*) & Schrittweite
  - Scannt Bild Zeilenweise
  - *Padding* legt Verfahren für Rand des Bildes fest
  - Ausgabe wird *activation* oder *feature map* genannt
- Praxis
  - *Convolutional Layer* mit 32 oder 16 Bit
  - Jeder Filter generiert eigene Ausgabematrix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

1. Bsp. Filter 2 x 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



Filter - Funktionsweise



Pooling Layer

- 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

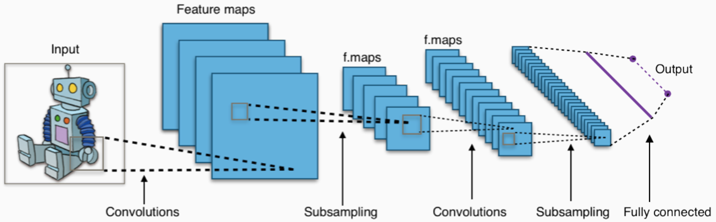
Pooling Layer

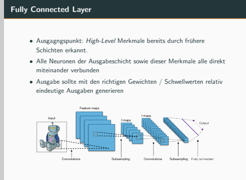
- Aggregiert die Ergebnisse von Convolutional Layern
- Ziele
  - Nur die relevantesten Signale an nächste Schicht weitergeben
  - Anzahl der Parameter im Netz reduzieren
- *MaxPooling Layer* am weitesten verbreitet

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

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





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

Deep Learning

Einführung - Thema 2

Silas Hoffmann

13. April 2020

Fachhochschule Wedel

Deep Learning

Einführung - Thema 2

Silas Hoffmann

13. April 2020

Fachhochschule Wedel

Alle Meterialen sind unter folgender URL zu finden:

https://github.com/derMacon/deeplearning_seminar

2020-04-13

Deep Learning

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

References i

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 Klassifizierungssproblem.
http://ekpwww.physik.uni-karlsruhe.de/~tkuhr/HauptseminarWS1112/Keck_handout.pdf.
Aufgerufen am: 15-03-2020.

References i

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 Klassifizierungssproblem.
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.

Deep Learning

2020-04-13

└ References

References ii

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.

- 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.

References iii
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.

References iv

- 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.

References iv
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.