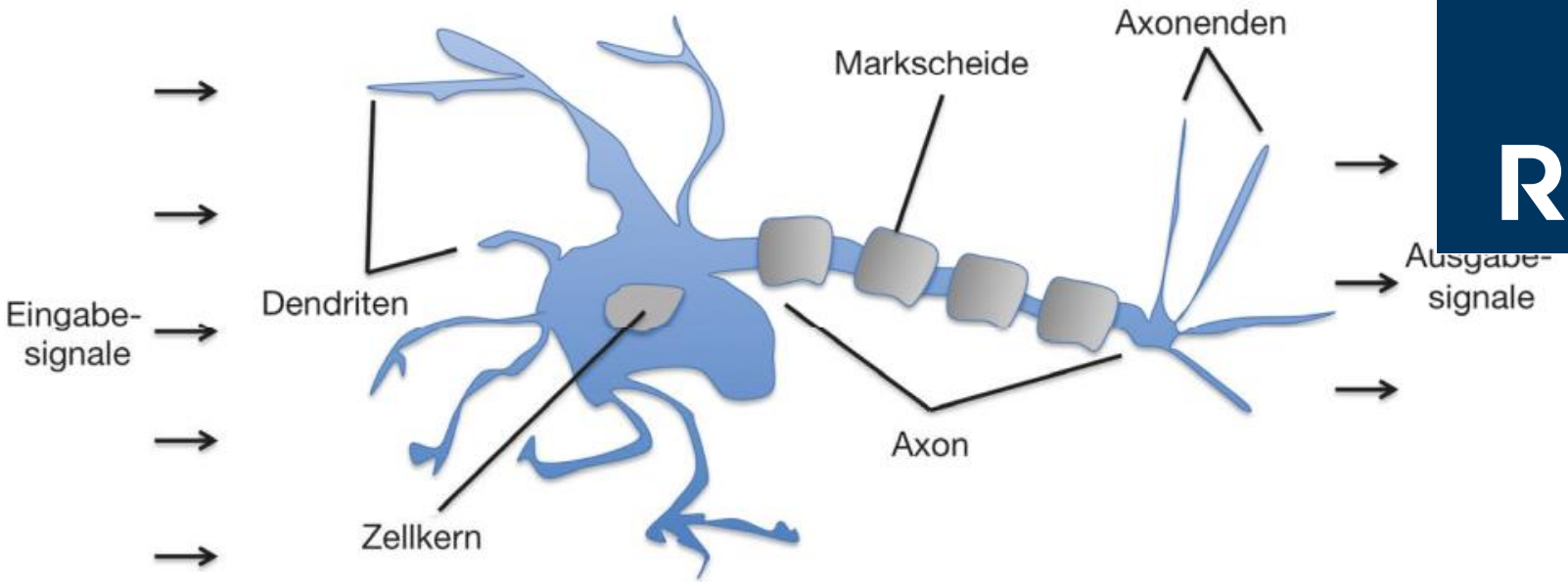


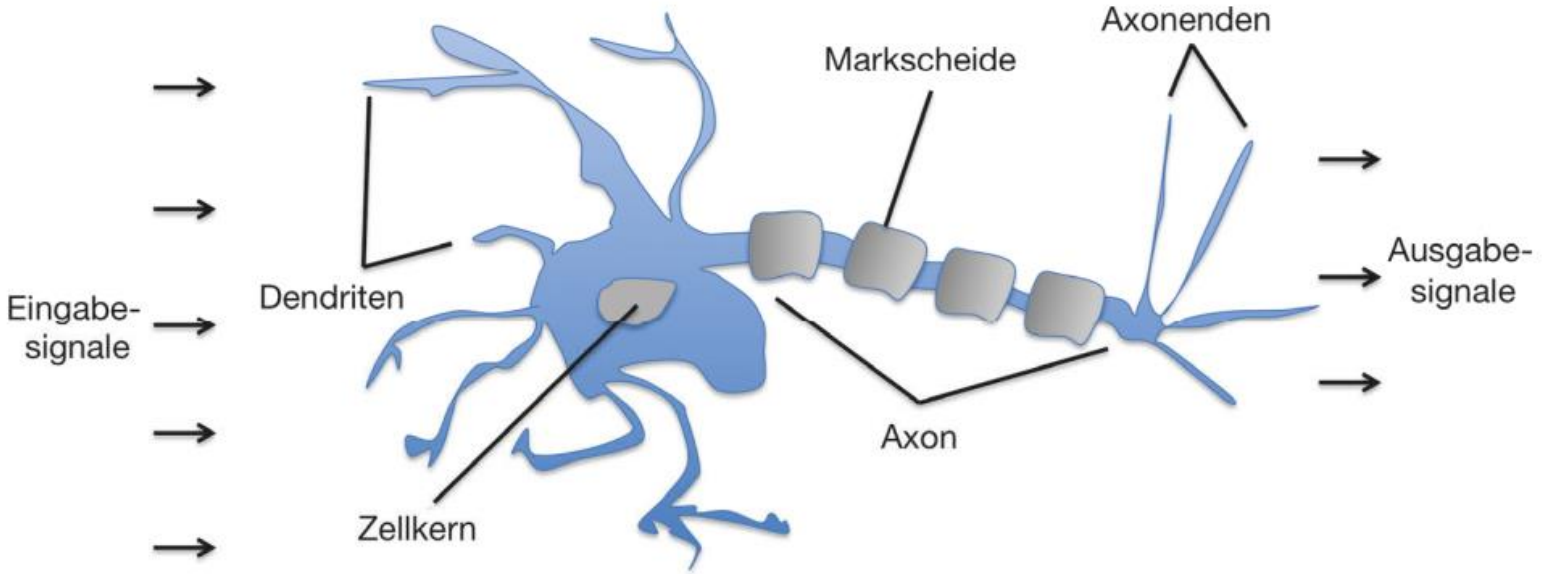
RUHR-UNIVERSITÄT BOCHUM

LERNALGORITHMEN

Training eines Klassifizierungsalgorithmus + Implementieren in Python

Agenda

- Biologischer Hintergrund
- Das **Perzeptron**
  - Grundbegriffe
    - Übersicht und Berechnungen
- Implementierung in Python
  - Interaktives Programmieren
- Adaptive lineare Neuronen
  - Gradientenabstiegsverfahren
    - Probleme
  - Stochastisches Gradientenabstiegsverfahren



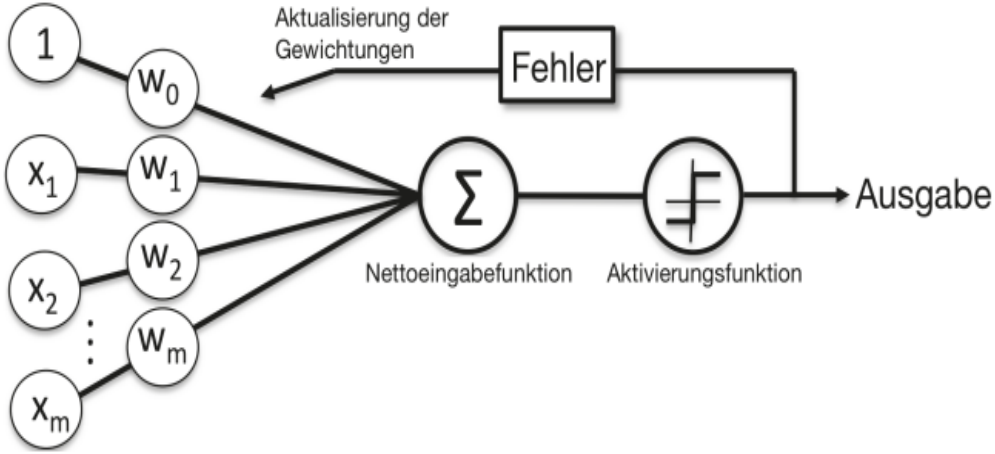
Dies ist der ungefähre Aufbau eines **Neurons** (also einer Zelle unseres Gehirns)

Ein Neuron kann bereits sortieren und aus viel Input wenig Ouput generieren.
Hierbei ist der Begriff **Schwellenwert** zum **Aktivieren** wichtig.

Idee: Programmieren eines ähnlichen Modells, welches wie das Gehirn Sortierungen durchführen kann

Das Perzeptron

Das Perzeptron



Übersicht über den Ablauf des Perzeptrons

Wir brauchen ein paar Definitionen:

- **Eingabevektor: x**
- **Gewichtungsvektor: w**
- **Nettoeingabe: z**
- **Aktivierungsfunktion: Φ(z)**
- **Ausgabewert: ŷ**

Grundbegriffe für das Perzeptron

Unser Perzeptron soll eine binäre Ausgabe haben

Dafür brauchen wir aber erstmal **Eingabevektor** und **Gewichtungsvektor**:

- Eingabevektor: $\boldsymbol{x}:=(x_1, x_2, \ldots, x_n)$
- Gewichtungsvektor: $\boldsymbol{w}:=(w_1, w_2, \ldots, w_n)$

Damit berechnen wir die **Nettoeingabe**.

- Nettoeingabe: $\boldsymbol{z}:=w^T x = w_1 * x_1 + w_2 * x_2 +, \ldots, +w_n * x_n$

Grundbegriffe für das Perzeptron

Aktivierungsfunktion: $\Phi(z)=\begin{cases} 1, wenn\ z \geq\ \theta \\ -1, andernfalls \end{cases}$

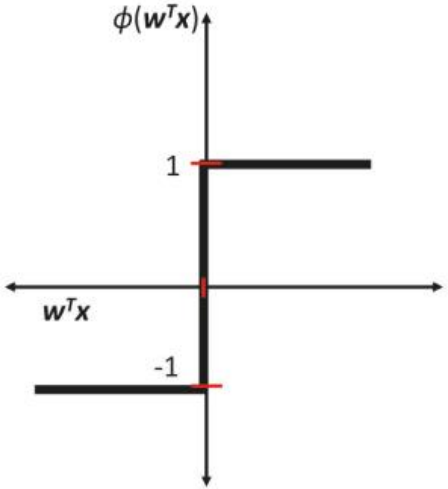
Der Einfachheit definieren wir das um und führen $w_0 = -\theta$ ein, sowie $x_0 = 1$

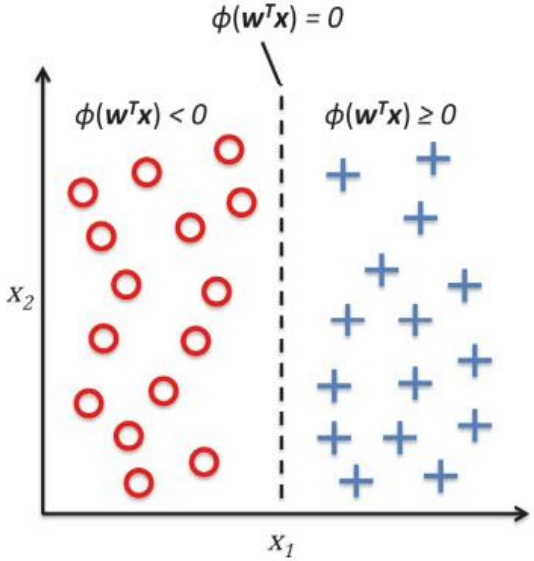
$\Rightarrow z = w_0 * x_0 + w_1 * x_1 + \cdots + w_n * x_n$

Sowie $\Phi(z)=\begin{cases} 1, wenn\ z \geq 0 \\ -1, andernfalls \end{cases}$

Später brauchen wir noch den **Ausgabewert** $\hat{y} = \Phi(z)$,

sowie $y =$ **Realwert**





Links:
Aktivierungsfunktion $\Phi(w^T x)$

Rechts:
Trennung von zwei Klassen, durch Anwendung von $\Phi(w^T x)$

Training des Perzeptrons

1. Die Gewichtungen werden mit 0 oder kleinen Werten initialisiert (später mit 0)
2. Für alle Trainigsobjekten x^i passiert folgendes
    1. Berechnung des Ausgabewerts ŷ
    2. Aktualisierung der Gewichtungen
        1. $w_j := w_j + \Delta w_j$
        2. $\Delta w_j = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)x_j^{(i)}$

Hierbei ist es wichtig zu erkennen, dass $\hat{y}^{(i)} > y^{(i)} \Rightarrow w_j(neu) < w_j(alt)$

Andersrum genauso.

Ein Problem was hierbei entsteht ist, wenn das Perzeptron die gegeben Daten nicht in zwei Klassen teilen kann (durch eine Hyperebene), so wird endlos versucht eine ideale Gewichtung zu finden.

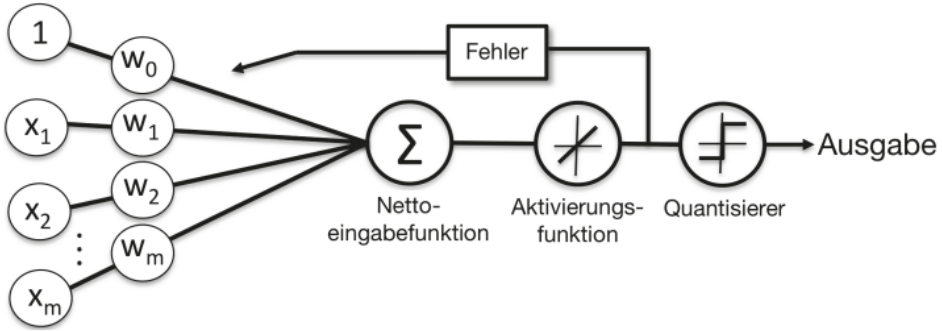
Das Perzeptron in Python

Adaline
Und das Gradientenabstiegsverfahren

Adaline

- Aktivierungsfunktion:
  lineare Funktion
- Aktivierungsfunktion Id
  anpassen durch Minimierung der Errorfunktion
  d.h. in diesem Fall $\Phi(w^t x) = w^t x$

- Im Anschluss wird ein Quantisierer benutzt der ähnlich ist wie die Sprungfunktion beim Perzeptron



Errorfunktion mit dem GD minimieren

Es ist notwendig eine bestimmte Zielfunktion zu definieren, diese wird oft als Errorfunktion (Straffunktion) bezeichnet. Diese soll minimiert werden.

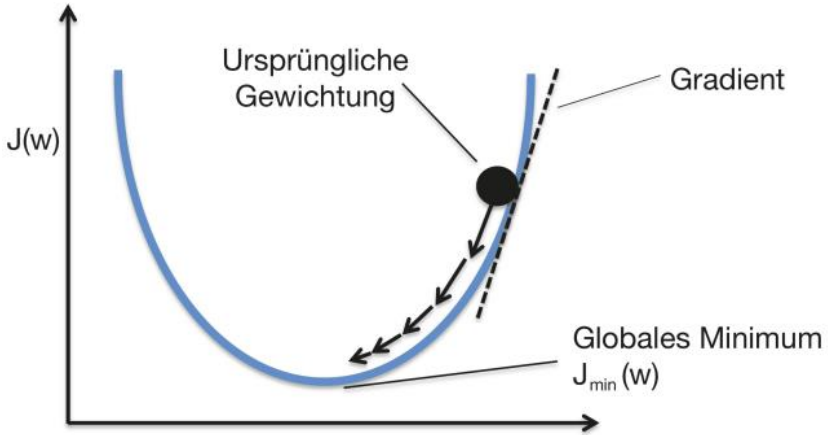
Bei Adeline betrachten wir die Errorfunktion J:

$$J(w) = \frac{1}{2}\sum_{i}\left(y^{(i)} - \Phi\left(z^{(i)}\right)\right)^2$$

Also die quadrierte Abweichung (MSE)

Diese Zielfunktion hat den praktischen Vorteil konvex zu sein

Gradientenabstiegsverfahren



Wir aktualisieren nun die Gewichtungen so, dass wir uns einen Schritt vom Gradienten entfernen, also in Richtung Minimum gehen

Die Gewichtungen werden nun so aktualisiert:

$$w := w + \Delta w$$

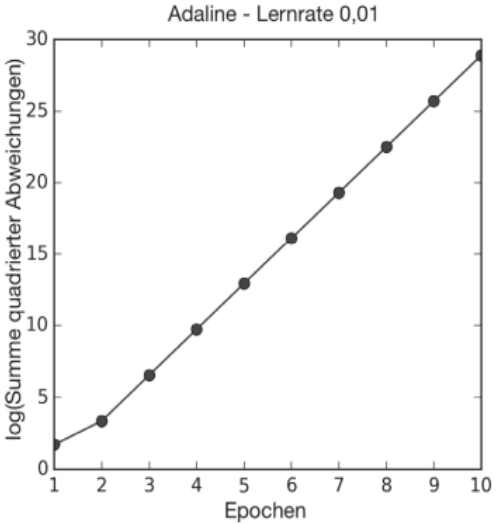
$$\Delta w := -\eta \nabla J(w)$$

$Um\ \nabla J$ zu berechnen brauchen wir die partiellen Ableitungen der Errorfunktion

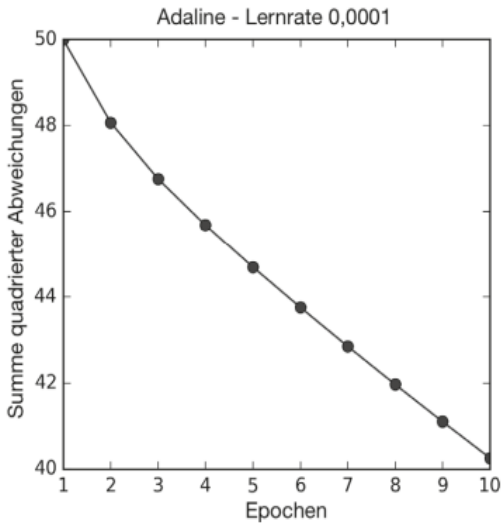
$$\frac{\partial J}{\partial w_j} = -\sum_i \left(y^{(i)} - \Phi\left(z^{(i)}\right)\right) x_j^{(i)}$$

$$\Rightarrow\ \Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i \left(y^{(i)} - \Phi\left(z^{(i)}\right)\right) x_j^{(i)}$$

Gradientenabstiegsverfahren



Links Adeline Abweichungen pro Epoche bei einer konstanten Lernrate von 0,01



Rechts Adeline Abweichungen pro Epoche bei einer konstanten Lernrate von 0,0001

Beide Lernraten bringen keine ersichtliche Konvergenz.

- Links schießt übers Ziel hinaus
- Rechts erreicht das Ziel „nie“ bzw. sehr langsam also in keiner effizienten Zeitspanne

Problem beim Gradientenabstiegsverfahren

Wir haben nun gesehen wie das Gradientenabstiegsverfahren funktioniert

Bei großen Datenmengen, wie sie im Machine Learning benutzt werden gibt es hier ein Problem, nämlich den Rechenaufwand.

$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i \left(y^{(i)} - \Phi\left(z^{(i)}\right)\right) x_j^{(i)}$ eine Summe bei einer Riesigen Datenmenge bedeutet riesigen Rechenaufwand.

Und da kommt das stochastische Gradientenabstiegsverfahren zur Rettung

Large-scale Machine Learning **stochastisches Gradientenabstiegsverfahren**

Stochastisches Gradientenabstiegsverfahren

Anstatt von $\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i \left(y^{(i)} - \Phi\left(z^{(i)}\right)\right) x_j^{(i)}$

Betrachten wir die Aktualisierung jedes Trainingsobjekt x^i einzeln

$$\eta \left(y^{(i)} - \Phi\left(z^{(i)}\right)\right) x^{(i)}$$

Dies ist deutlich ungenauer als das GD aber auch sehr viel schneller auszuführen.

Es ist wichtig, dass die Trainigsobjekte, deren Aktualisierungen wir beobachten zufällig gewählt werden

Stochastisches Gradientenabstiegsverfahren

Oft wird auch die Lernrate angepasst, indem sie zu einer adaptiven Lernrate gemacht wird, d.h. einer Lernrate die am Anfang größer ist als am Ende

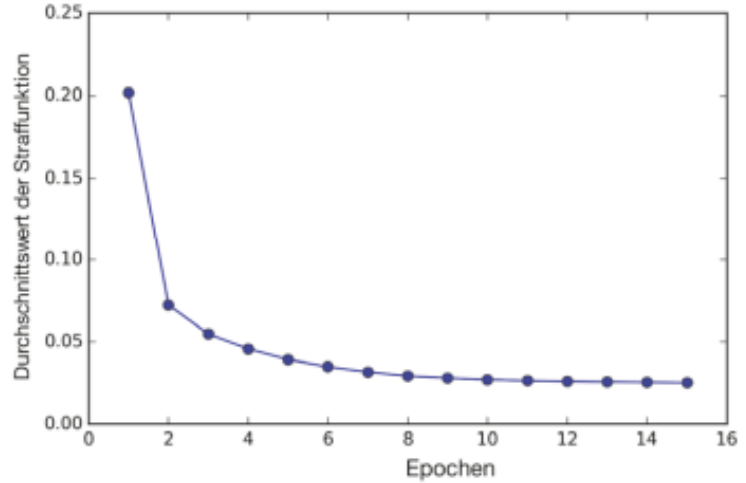
z.B. $\frac{c_1}{[Anzahl\ der\ Iterationen]+c_2}$ wobei c_1 und c_2 Konstanten sind

Ein Beispiel für eine adaptive Lernrate wäre auch Adam (Dieser ist eine Art ausgefeiltere Version einer adaptiven Lernrate)

Dass GSD wird auch oft beim *online Learning* verwendet, wo es um große in Echtzeit eintreffende Datenmengen geht (Werbung).

Stochastisches Gradientenabstiegsverfahren

- Im Bild rechts sieht man
  - Die Funktion scheint schnell zu konvergieren
- Das schnelle durcharbeiten der Epochen und dadurch das schnelle sinken des Durchschnittswertes der Errorfunktion ist der Grund warum das SGD beim Online Learning beliebt ist
- Es gibt auch noch einen Mittelweg zwischen den beiden Extremen, alle Werte nehmen und nur einen nehmen (*Mini-Batch-Learning*)
  - Man kann sich auch eine Anzahl an Werten z.B. 50 auf einmal angucken



Errorfunktion bei stochastischem Gradientenabstiegsverfahren

Vielen Dank fürs Zuhören

Falls es noch Fragen gibt gerne stellen!

Ansonsten Frohe Weinachten!

Quellen

BildQuelle:

https://www.publicdomainpictures.net/de/view-image.php?image=28179&picture=frohe-weihnachten-4

TextQuelle:

RaschkaSebastia_2017_Kapitel2Lernalgorithm_MachineLearningMitPyt