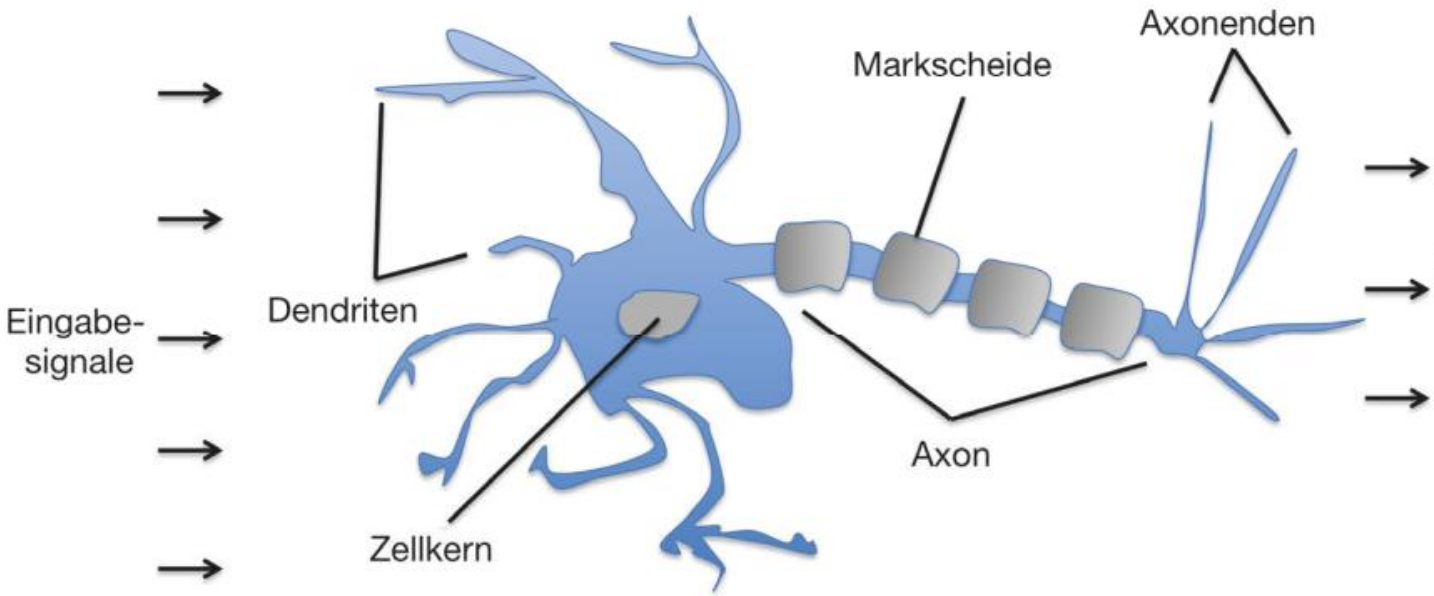


RUHR-UNIVERSITÄT BOCHUM

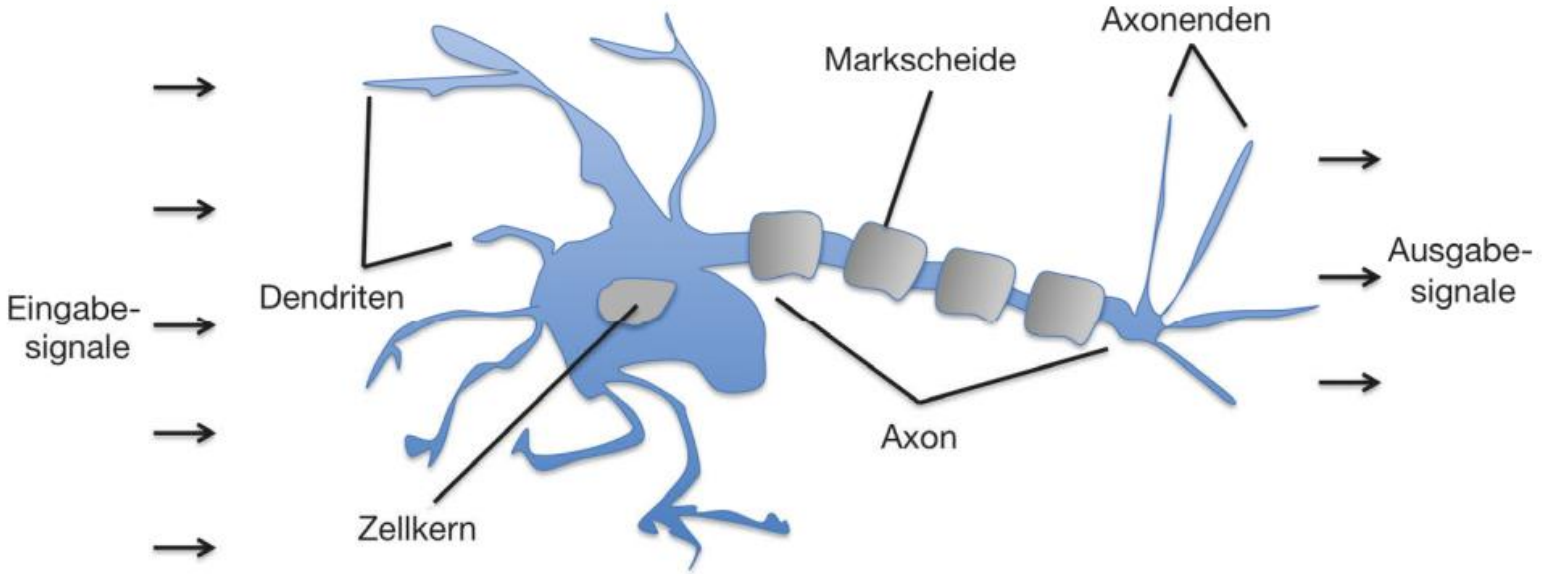
LERNALGORITHMEN

Training eines Klassifizierungsalgorithmus + Implementieren in Python

Agenda

- Biologischer Hintergrund
- Das **Perzeptron**
  - Grundbegriffe
  - Perzeptron in Python
- **Adaline** (lineare Neuronen)
  - **Gradientenabstiegsverfahren (GD)**
  - Adaline in Python mit GD
  - Problem mit GD
  - **Stochastisches Gradientenabstiegsverfahren (SGD)**

RUHR UNIVERSITÄT BOCHUM RUB



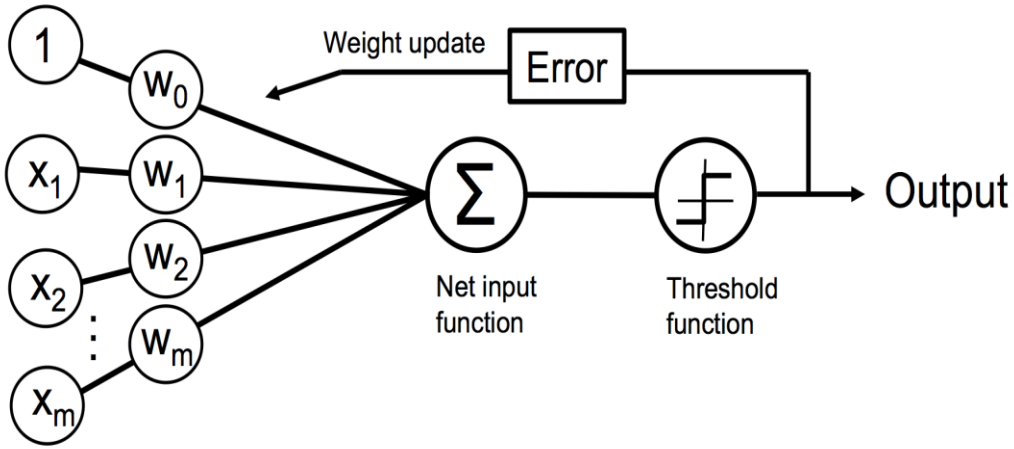
Dies ist der ungefähre Aufbau eines **Neurons** (also einer Zelle unseres Gehirns)

Neuron kann sortieren und selektieren
Wichtige Begriffe **Schwellenwert** und **Aktivieren**

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

Das Perzeptron

Das Perzeptron



Neuronales Netz mit nur einem Layer

Wir brauchen ein paar Definitionen:

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

RUHR UNIVERSITÄT BOCHUM RUB

Grundbegriffe für das Perzeptron

Unser Perzeptron soll eine binäre Ausgabe haben

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

RUHR UNIVERSITÄT BOCHUM RUB

Grundbegriffe für das Perzeptron

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

Nullgewichtung dadurch führen wir $w_0 = -\theta$ ein, sowie $x_0 = 1$

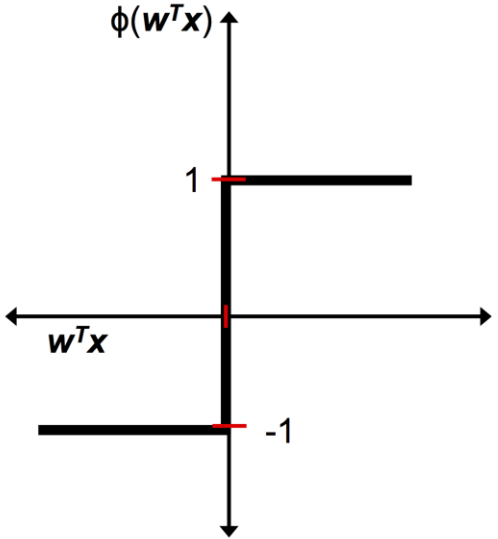
$\Rightarrow z = w_0 * x_0 + w_1 * x_1 + \cdots + w_n * x_n$

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

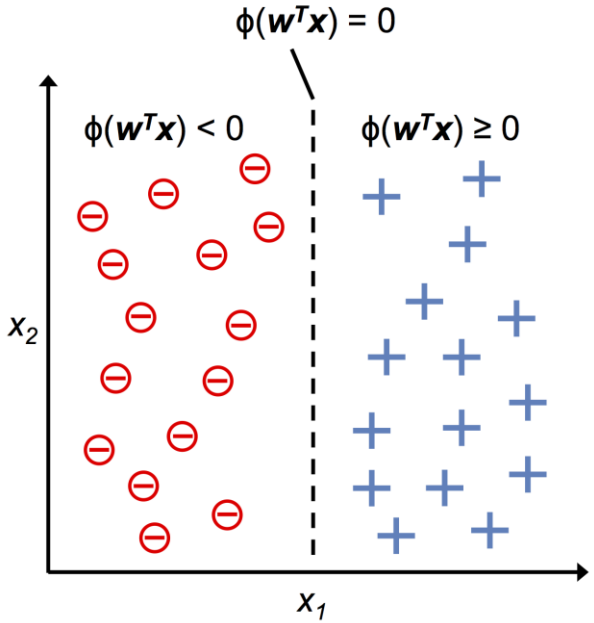
Später brauchen wir noch den **Ausgabewert** $\Phi(z) = \hat{y}$,

sowie $y =$ **Realwert**

RUHR UNIVERSITÄT BOCHUM RUB



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



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

RUHR UNIVERSITÄT BOCHUM RUB

Training des Perzeptrons

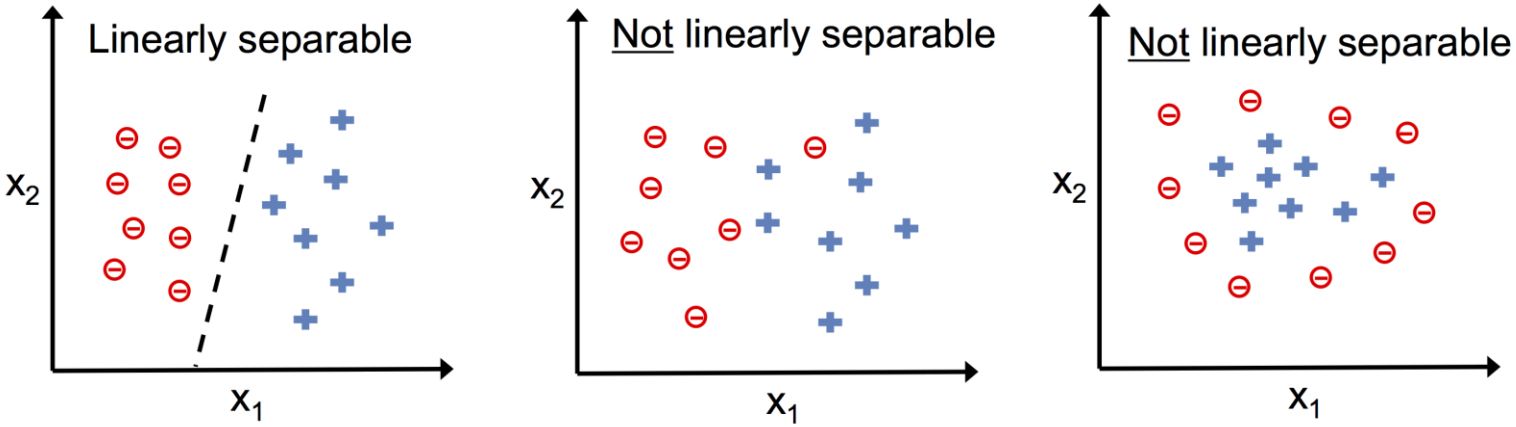
1. Die Gewichtungen werden mit 0 oder kleinen Werten initialisiert (später mit 0)
2. Für alle Trainigsobjekten x^i passiert folgendes
    1. Berechnung des Ausgabewerts ŷ
    2. Aktualisierung der Gewichtungen
        1. $w_j := w_j + \Delta w_j$
        2. $\Delta w_j = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)x_j^{(i)}$

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

Andersrum genauso.

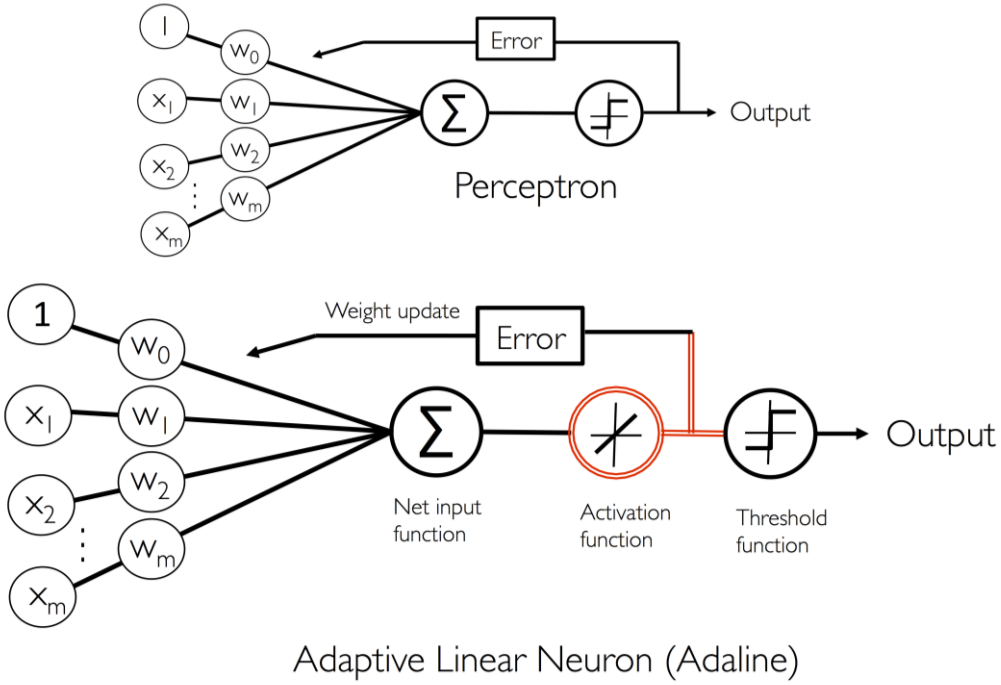
RUHR UNIVERSITÄT BOCHUM RUB

Problem mit dem Perzeptron



Das Perzeptron in Python

Adaline
Und das Gradientenabstiegsverfahren



Perzeptron im Vergleich mit Adeline

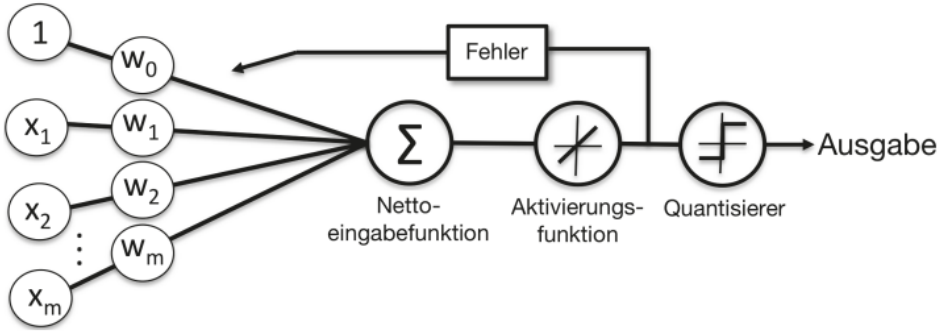
RUHR UNIVERSITÄT BOCHUM RUB

Adaline

- Aktivierungsfunktion:
  lineare Funktion
- Aktivierungsfunktion beginnt als Identitäts Abbildung

$$\Phi(w^t x) = w^t x$$

- Gewichts Anpassung durch Minimierung der Errorfunktion
- Quantisierer sorgt wieder für Binären Output



RUHR
UNIVERSITÄT
BOCHUM
RUB

Errorfunktion mit dem GD minimieren

Bestimmung einer Zielfunktion, diese wird oft als Errorfunktion (Straffunktion) bezeichnet.

Bei Adeline betrachten wir die Errorfunktion J:

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

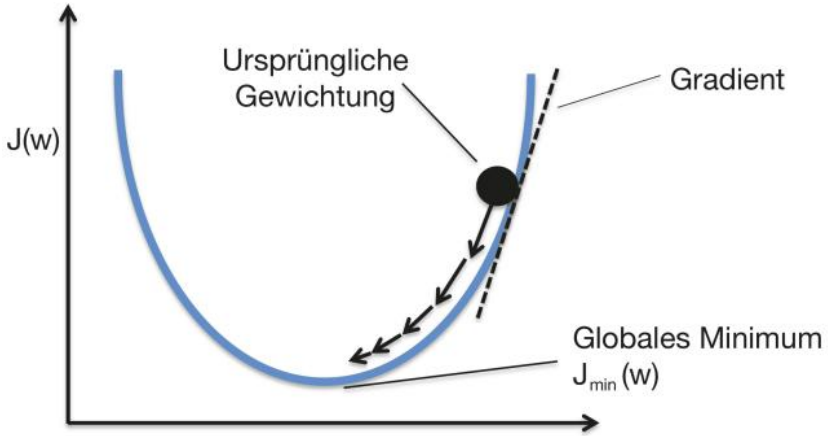
Also die quadrierte Abweichung (MSE)

Diese Zielfunktion hat den praktischen Vorteil konvex zu sein

⇒ hat Minimum

RUHR UNIVERSITÄT BOCHUM RUB

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

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

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

RUHR UNIVERSITÄT BOCHUM RUB

Perzeptron Adeline Unterschiede

- Perzeptron hat ganzzahlige Ausgabe von $\Phi(z)$
- Perzeptron Gewichtaktualisierung nach jedem Objekt

$$w_j := w_j + \Delta w_j$$

$$w_0 = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)$$

$$\Delta w_{j\in[1,n]} = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)x_j^{(i)}$$

- Adeline hat eine reele Zahl als Ausgabe von $\Phi(w^t x)$
- Adeline Gewichtaktualisierung beruht auf allen Objekten

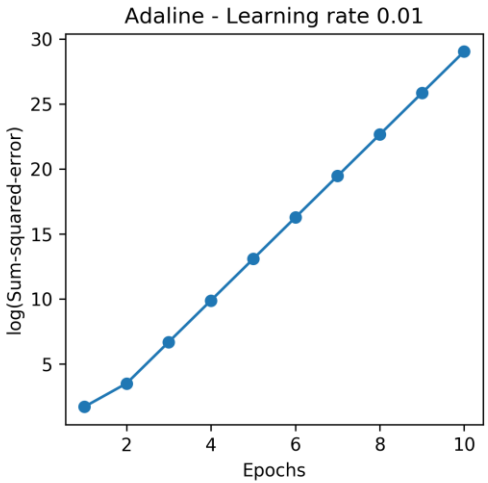
$$w_0 = \eta * \sum errors, \text{ wobei } errors = (y - X^T * w)$$

$$w_{i\in[1,n]} = \eta * X^T * (y^{(i)} - X^T * w^{(i)})$$

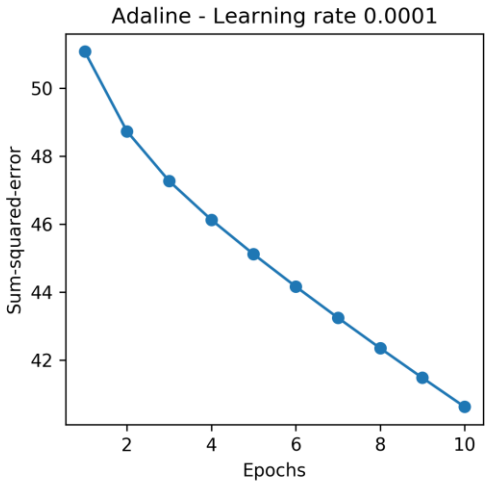
- Wie bei Perzeptron Sammlung der Fehler um Konvergenz zu erkennen
  - Diesmal Cost genannt in self.cost_ gespeichert

RUHR UNIVERSITÄT BOCHUM RUB

Gradientenabstiegsverfahren



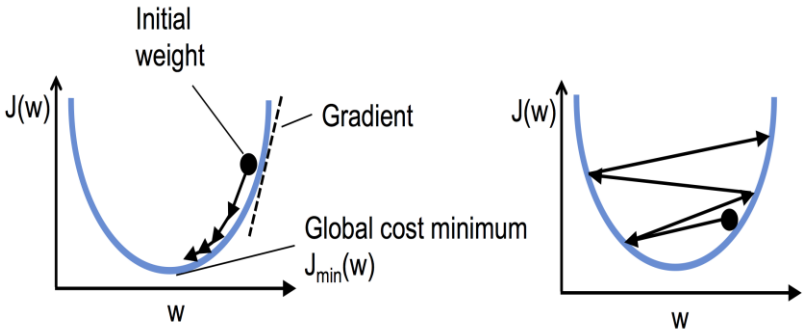
Links Adeline Abweichungen pro Epoche bei einer konstanten Lernrate von 0,01



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

Beide Lernraten bringen keine ersichtliche Konvergenz.

- Rechts Lernrate zu groß
- Links Lernrate zu klein



RUHR UNIVERSITÄT BOCHUM RUB

Adeline in Python mit Gradientenabstiegsverfahren

Problem beim Gradientenabstiegsverfahren

Wir haben nun gesehen wie das Gradientenabstiegsverfahren funktioniert

Bei großen Datenmengen ⇒ großer Rechenaufwand!

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

Summe sehr groß für viele i ⇒ eine Iteration dauert sehr lang

Und da kommt das stochastische Gradientenabstiegsverfahren zur Rettung

RUHR UNIVERSITÄT BOCHUM RUB

Large-scale Machine Learning **stochastisches Gradientenabstiegsverfahren**

Stochastisches Gradientenabstiegsverfahren

Anstatt Gewichtung mit allen Testobjekten

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

Benutzen für die Aktualisierung nur die Abweichung eines zufälligen Trainingsobjekt x^i

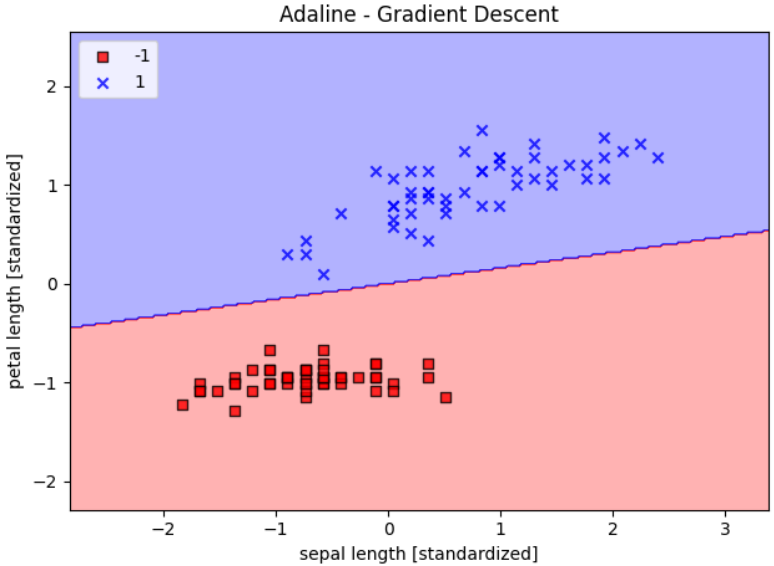
$$\eta \left(y^{(i)} - \Phi\left(z^{(i)}\right)\right) x^{(i)}$$

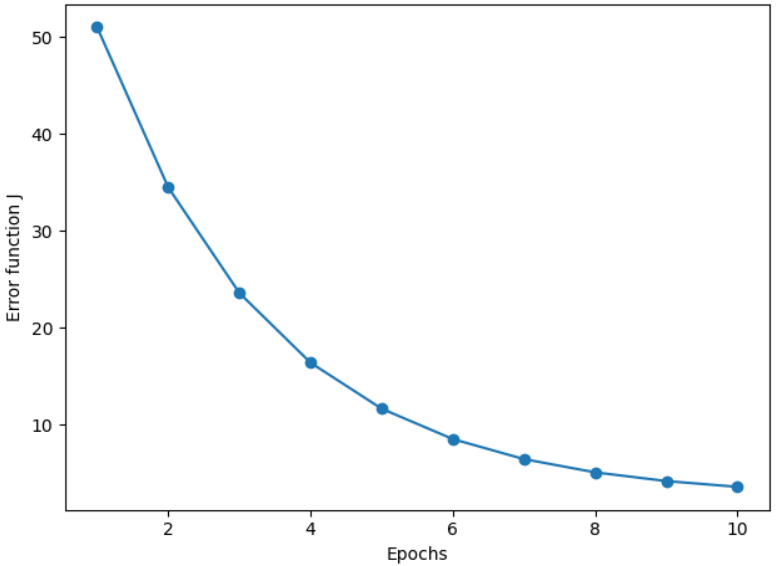
Deutlich ungenauer als GD, aber auch deutlich schnellere Iterationen

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

RUHR UNIVERSITÄT BOCHUM RUB

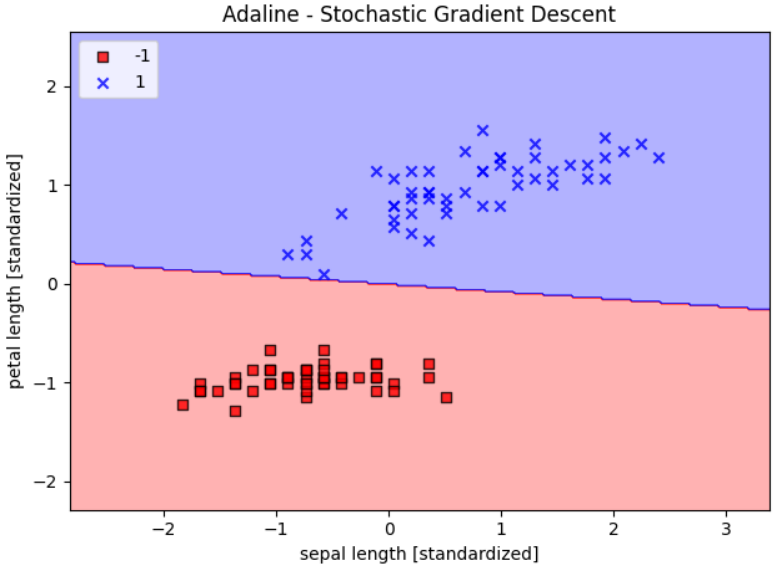
Gradientenabstiegsverfahren

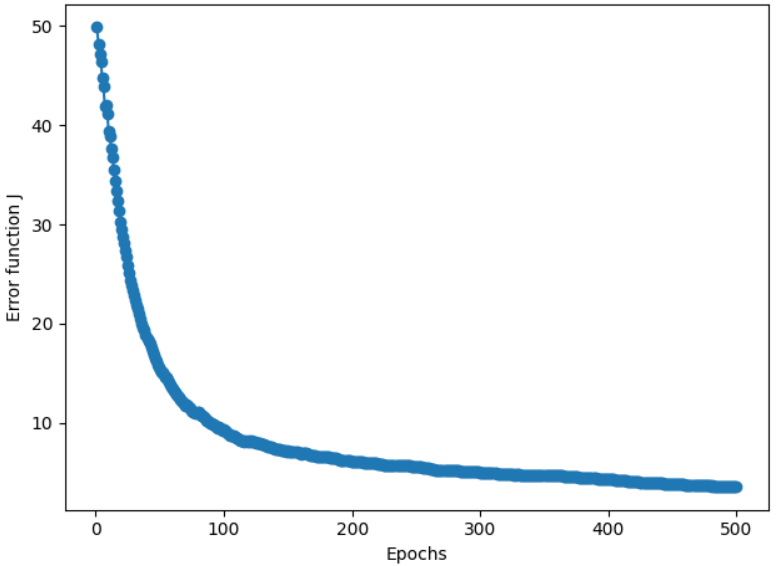




RUHR UNIVERSITÄT BOCHUM RUB

Stochastisches Gradientenabstiegsverfahren





RUHR UNIVERSITÄT BOCHUM RUB

Stochastisches Gradientenabstiegsverfahren

Um das stochastische Gradientenabstiegsverfahren genauer zu machen wird oft eine **adaptive Lernrate** benutzt

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

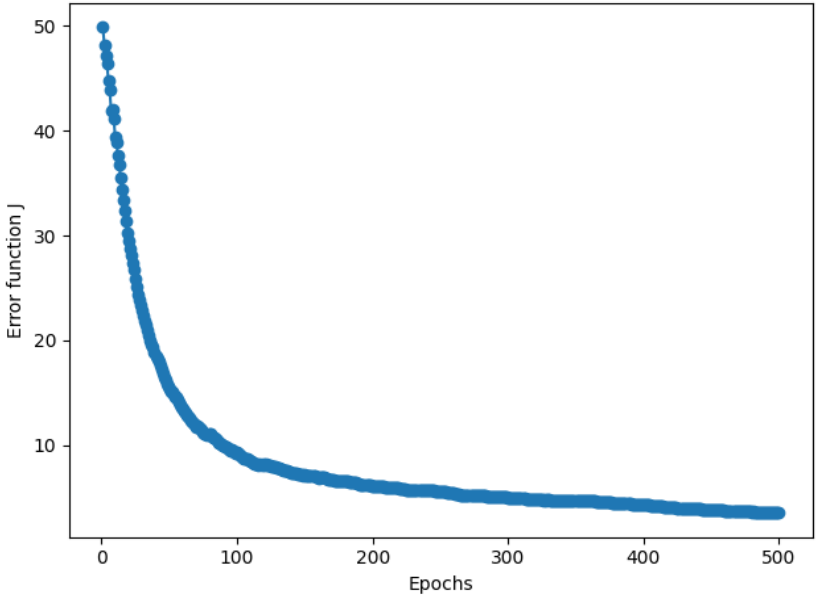
Ein Beispiel für eine adaptive Lernrate wäre auch **Adam** (Dieser ist eine ausgefeilte Version einer adaptiven Lernrate)

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

RUHR UNIVERSITÄT BOCHUM RUB

Stochastisches Gradientenabstiegsverfahren

- Schnelle Iterationen sind gut bei großen Datenmengen
  - Daher beliebt bei Online Learning
  - Ungenauer als GD
- Mittelweg zwischen GD und SGD (*Mini-Batch-Learning*)
  - Hierbei wendet man das GD auf kleinere Teilmengen an



Errorfunktion bei stochastischem Gradientenabstiegsverfahren

RUHR UNIVERSITÄT BOCHUM RUB

Vielen Dank fürs Zuhören

Falls es noch Fragen gibt gerne stellen!

Frohe Weinachten!

Quellen

BildQuellen:

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

RaschkaSebastian Machine Learning mit Python von 2017

TextQuelle:

RaschkaSebastian Machine Learning mit Python von 2017