Technische Hochschule Ingolstadt

Einsatz von Gradientenabstiegsverfahren in Neuronalen Netzen

Dr. Cristian Axenie

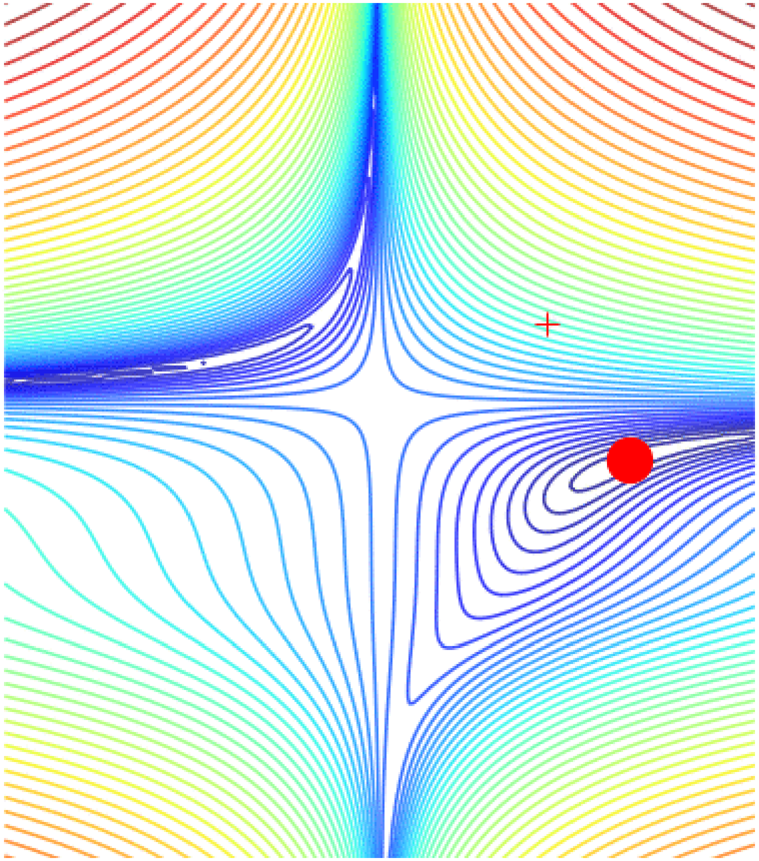
Inhalt

Grundlagen der Optimierung

Grundlagen der Optimierung

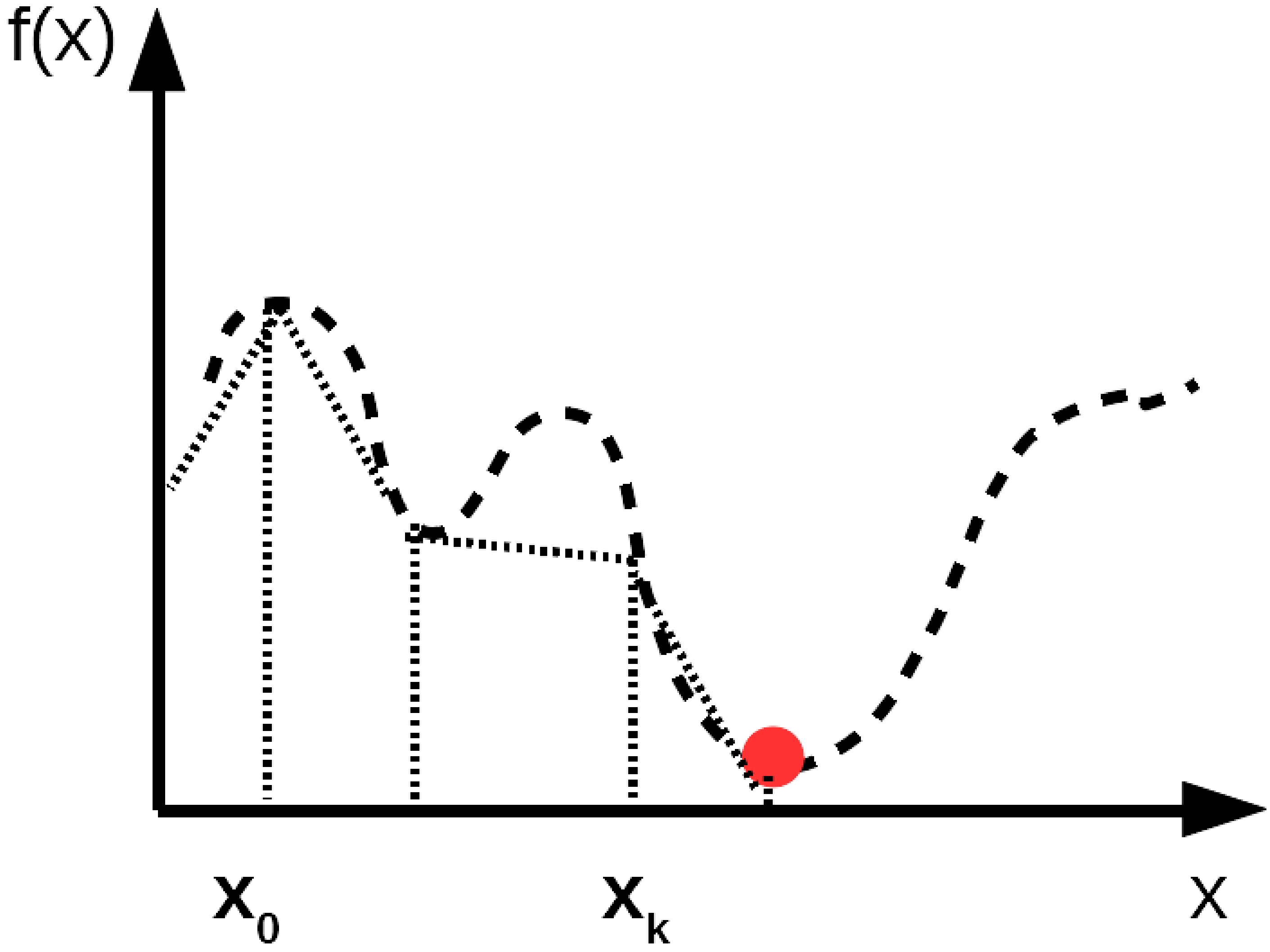
Optimierungsalgorithmen sind in der Regel **iterative Verfahren**.

Ausgehend von einem gegebenen Punkt x_0+ erzeugen sie eine Folge x_k von **Iterierten**, die zu einer Lösung (●) **konvergieren** [2].

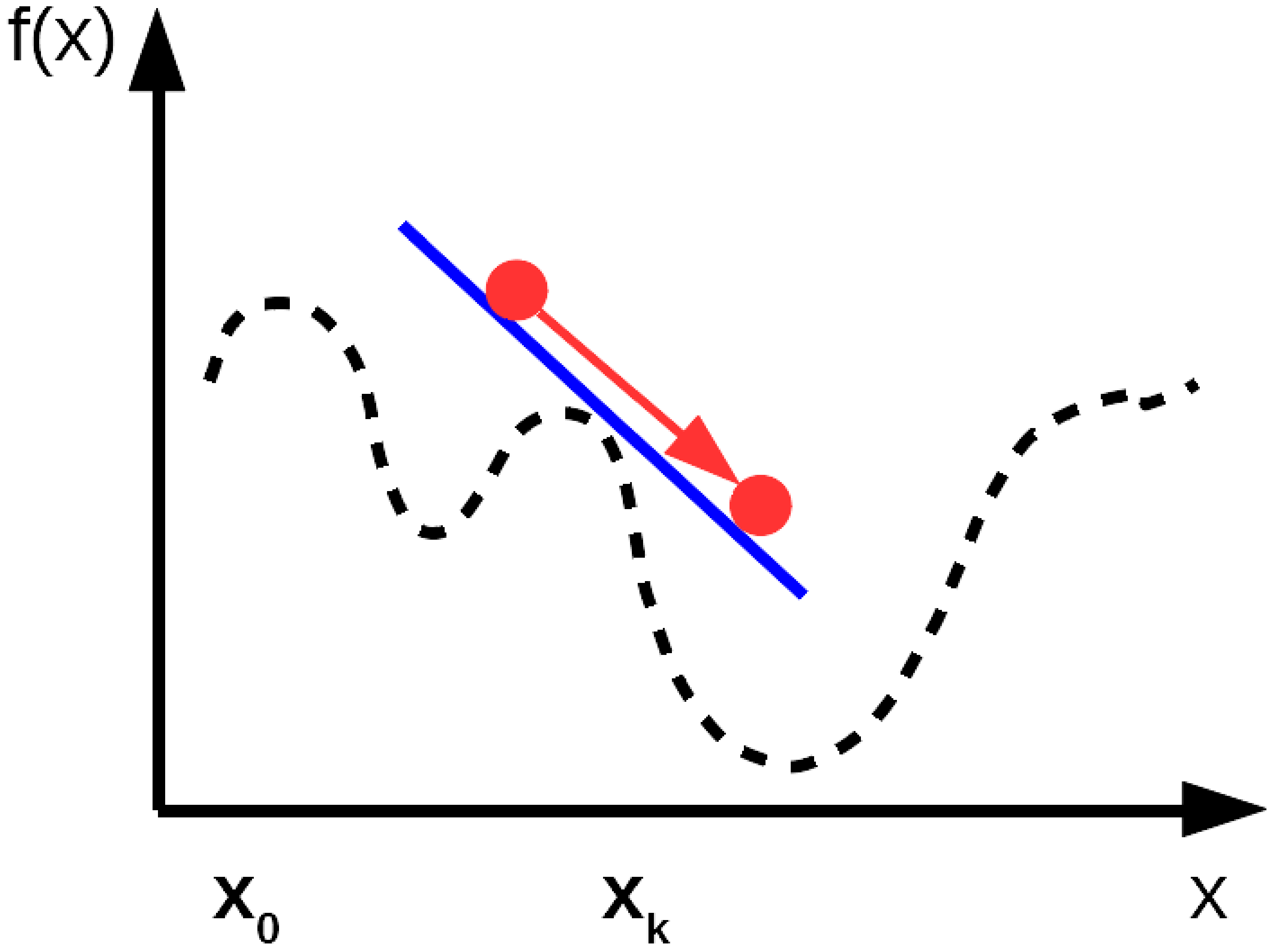


Grundlagen der Optimierung

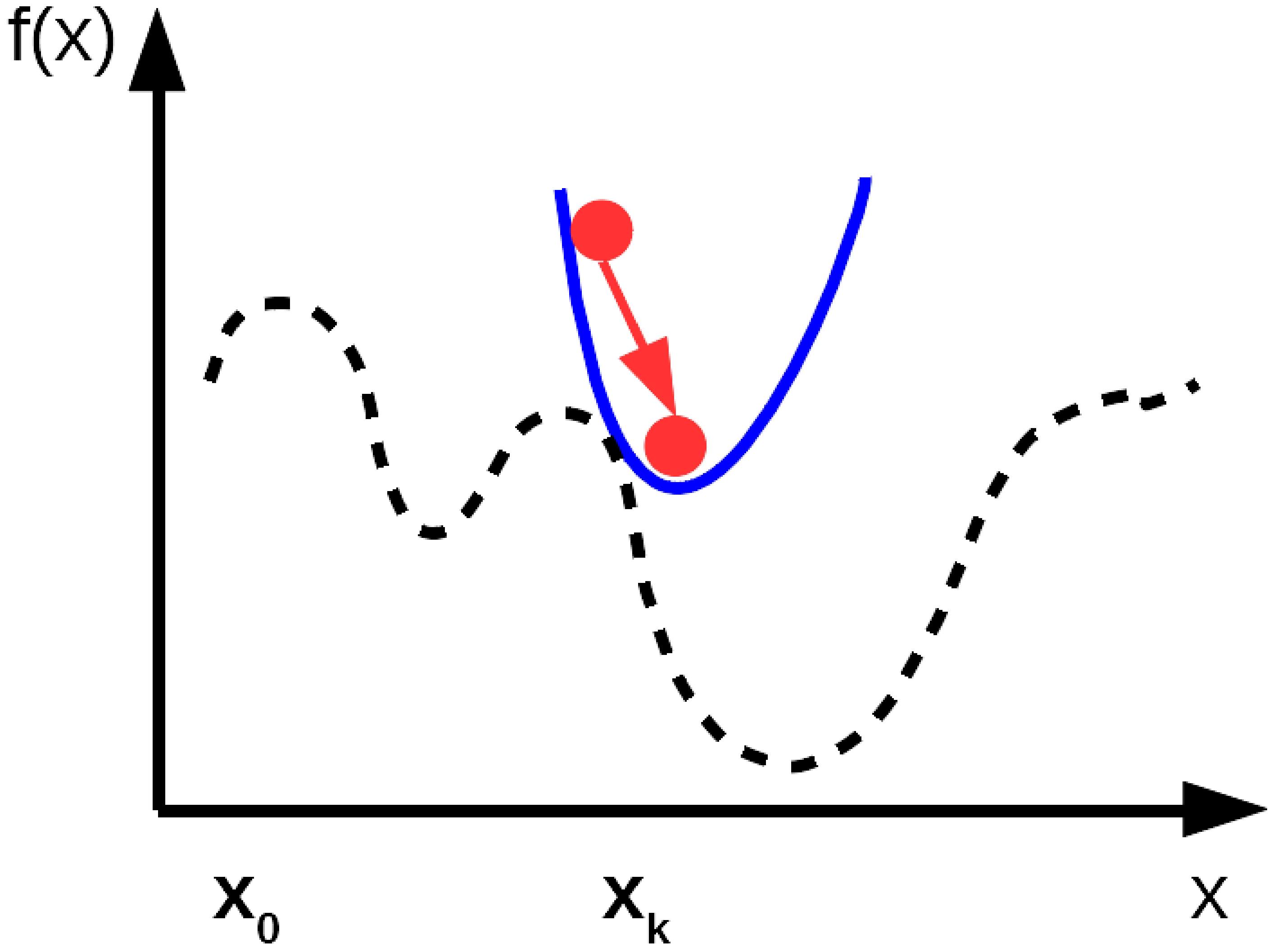
Optimierung nullter Ordnung



Optimierung erster Ordnung

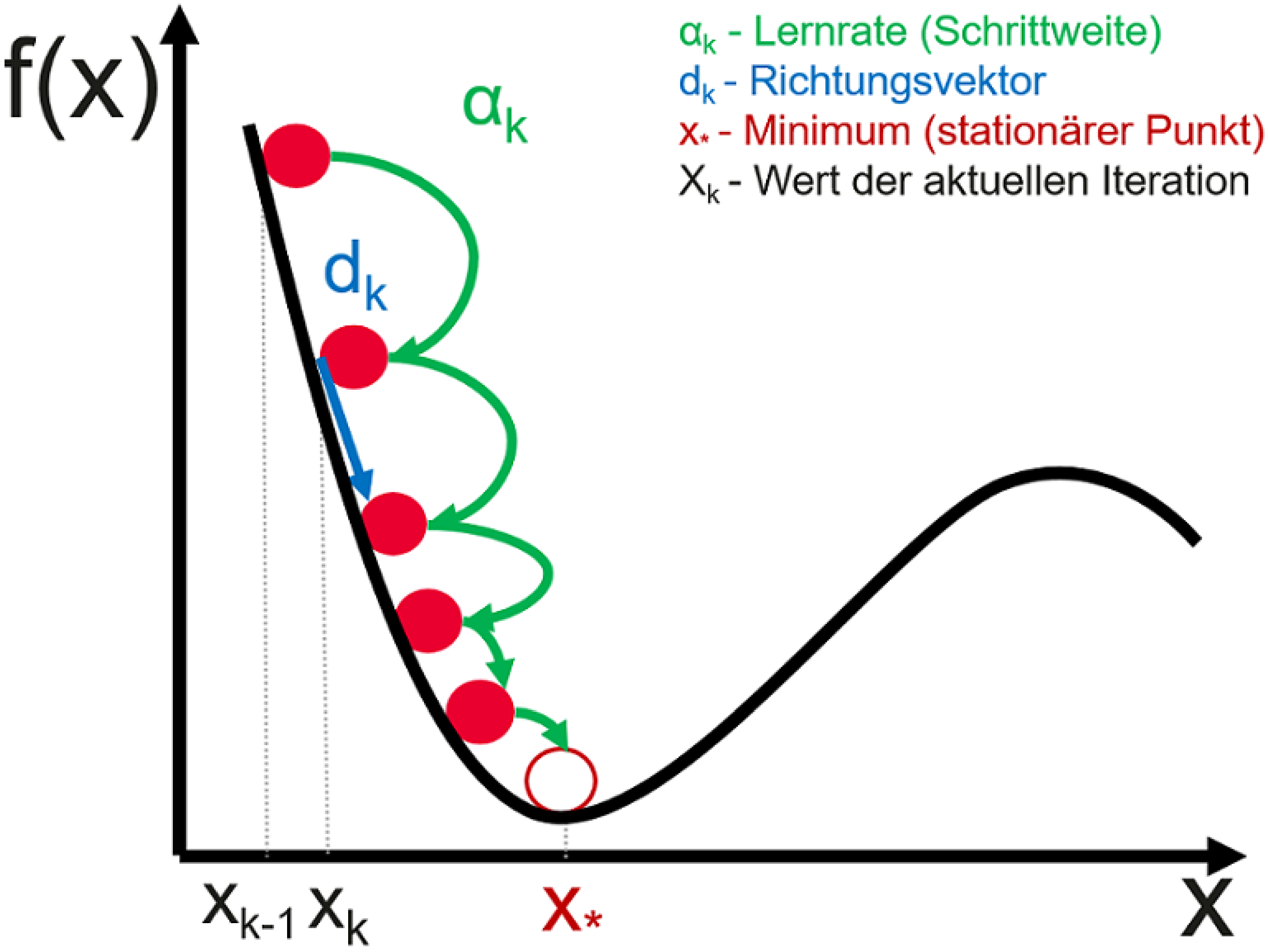


Optimierung zweiter Ordnung



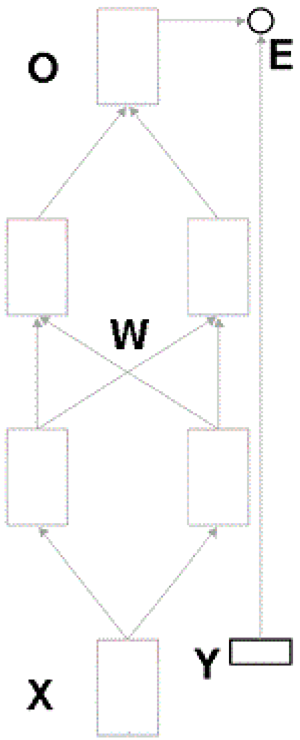
Gradientenabstieg

Gradientenabstieg



Lernen als Optimierung in Neuronale Netzen

Lernen als Optimierung in Neuronale Netzen

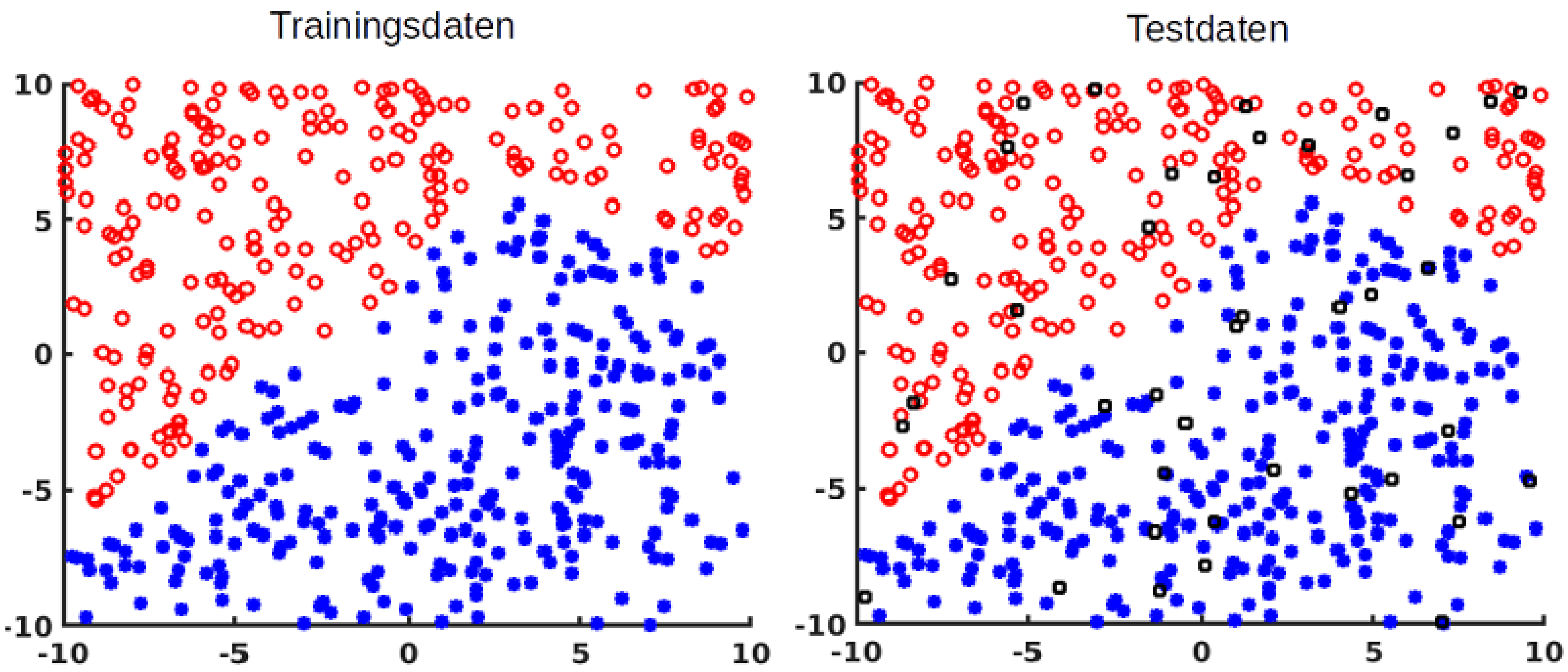


Optimierungsalgorithmen in **neuronalen Netzen**[3]:

- **Zielfunktion zu minimieren (E)**
- **internen lernbaren Parametern (W)**
- **erwarteten Werte (Y)**
- **Prädiktoren (X)**
- **tatsächlichen Netzwerkausgangswert (O)**

Gradientenabstieg in Neuronale Netzen

Binäre Klassifikation mit neuronalen Netzen

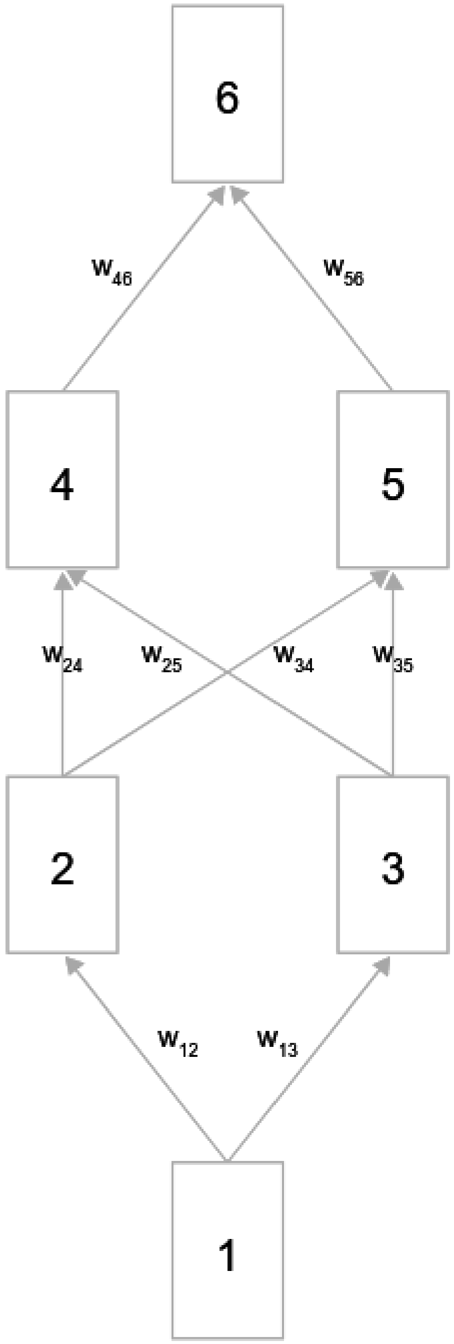


O – Klasse 1
■ – Klasse 2
O – Testpunkte

Neuronale Netzwerkstruktur

Neuronales Netzwerk zur Lösung der binären Klassifizierungsaufgabe.

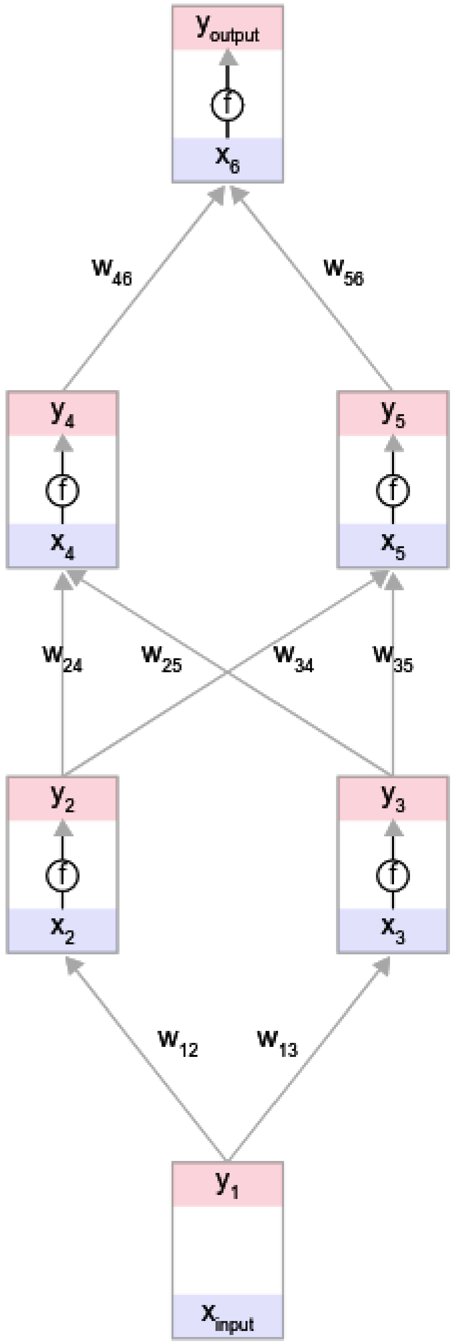
Neuronen in verknüpften Schichten sind über Kanten mit Gewichten w_{ij} verbunden (d.h. welche sind die Netzwerkparameter).



Aktivierungsfunktion

Jedes Neuron hat eine Gesamteingabe x , eine Aktivierungsfunktion $f(x)$ und eine Ausgabe $y = f(x)$.

$f(x)$ muss eine nichtlineare Funktion sein (z. B. ReLu, tanh, sigmoid).

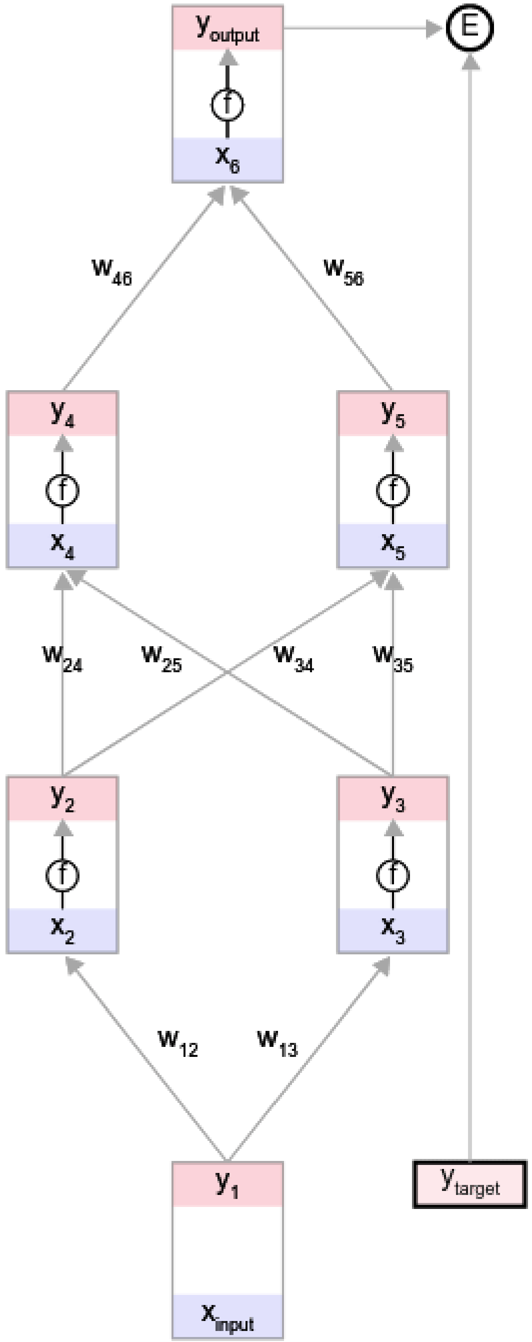


Fehler- / Verlustfunktion

Das Netzwerk lernt die Gewichte, so dass für alle Eingänge x_{input} die vorhergesagte Ausgabe y_{output} nah an y_{target} ist.

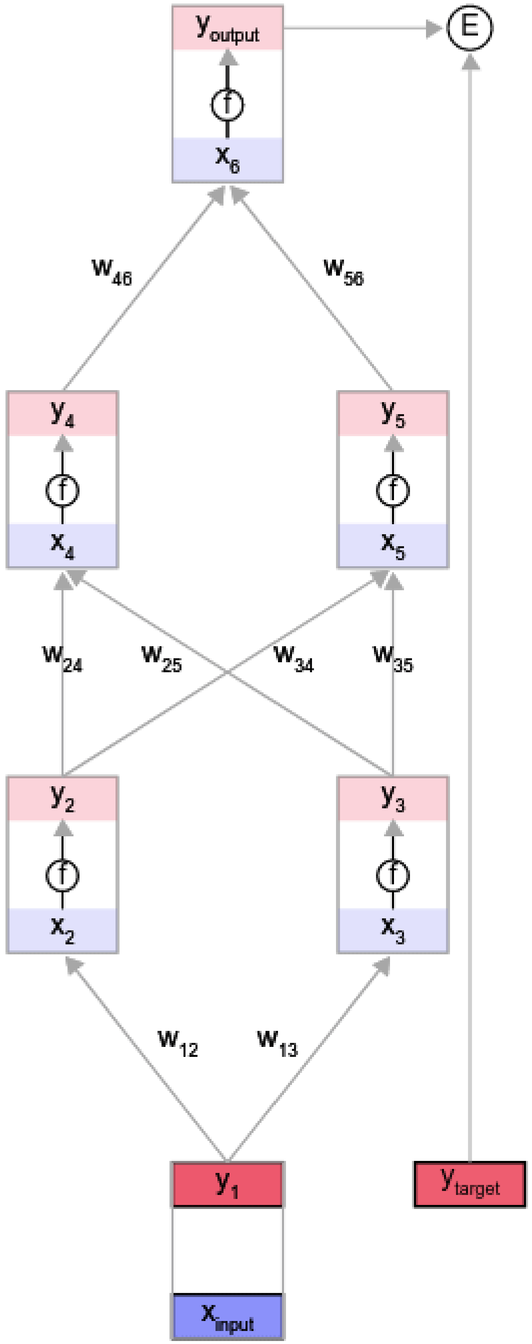
Fehlerfunktion ist die mittlere quadratische Fehler:

$$E(y_{output}, y_{target}) = \frac{1}{2}(y_{output} - y_{target})^2$$



Forwardpropagation

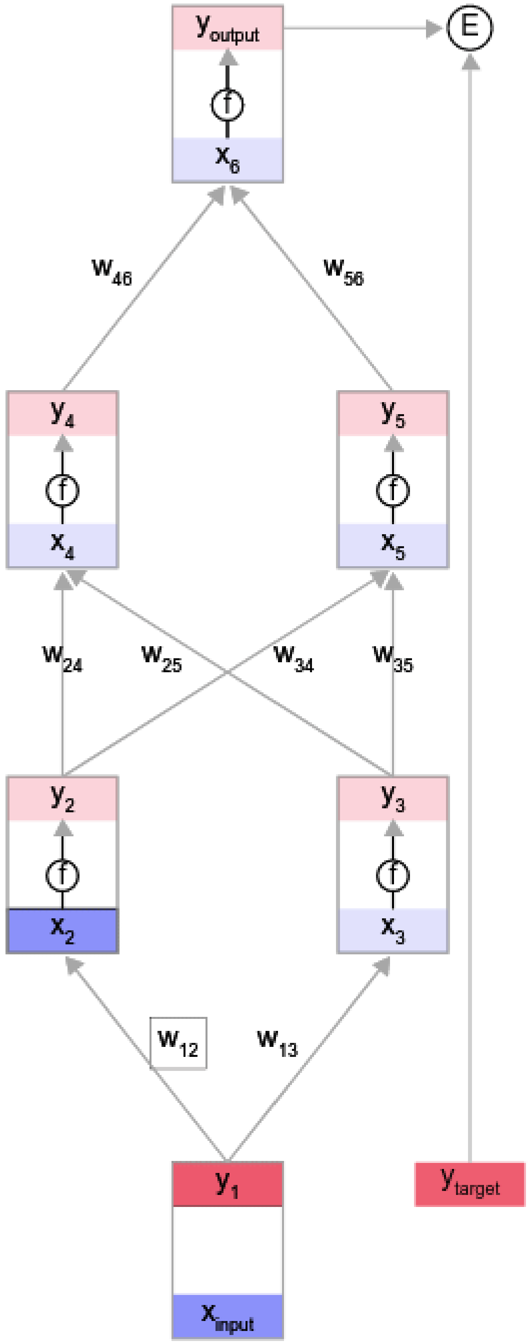
Das Netzwerk verwendet Eingabebeispiele (x_{input}, y_{target}), um alle Neuronen zu aktualisieren.



Forwardpropagation

Um die verborgenen Schichten zu aktualisieren, verwendet das Netzwerk die Ausgabe y der vorherigen Schicht und berechnet mit den Gewichten die Eingabe x der Neuronen in der nächsten Schicht:

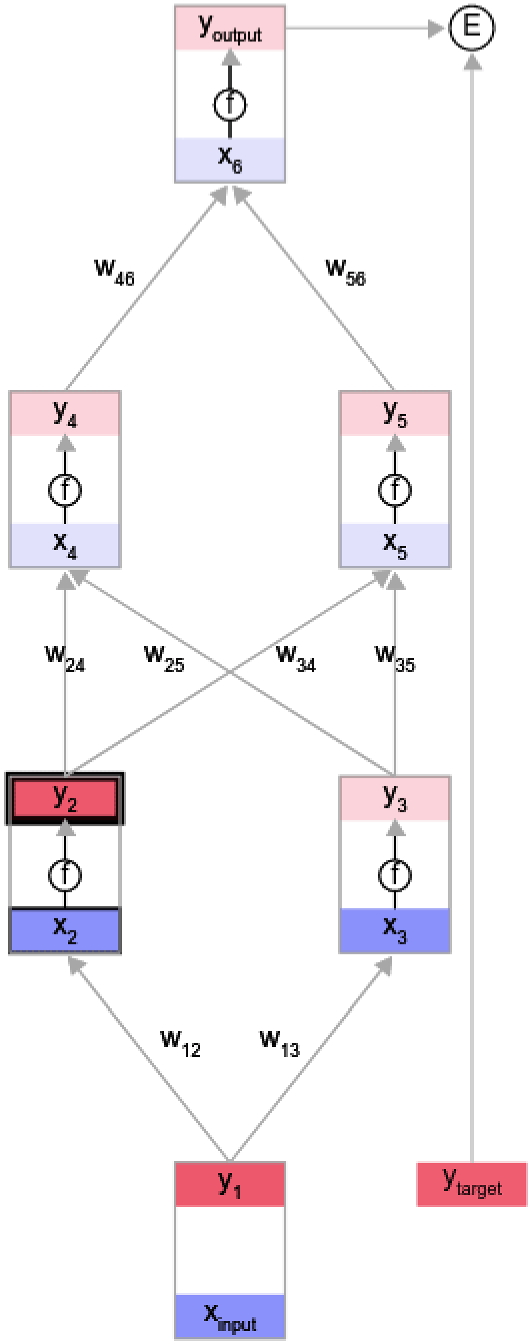
$$x_j = \sum_{i \in in(j)} w_{ij} y_i + b_j$$



Forwardpropagation

Das Netzwerk aktualisiert die Ausgabe der Neuronen in den verborgenen Schichten durch die nichtlineare Aktivierungsfunktion, $f(x)$.

$$y = f(x) = tanh(x)$$

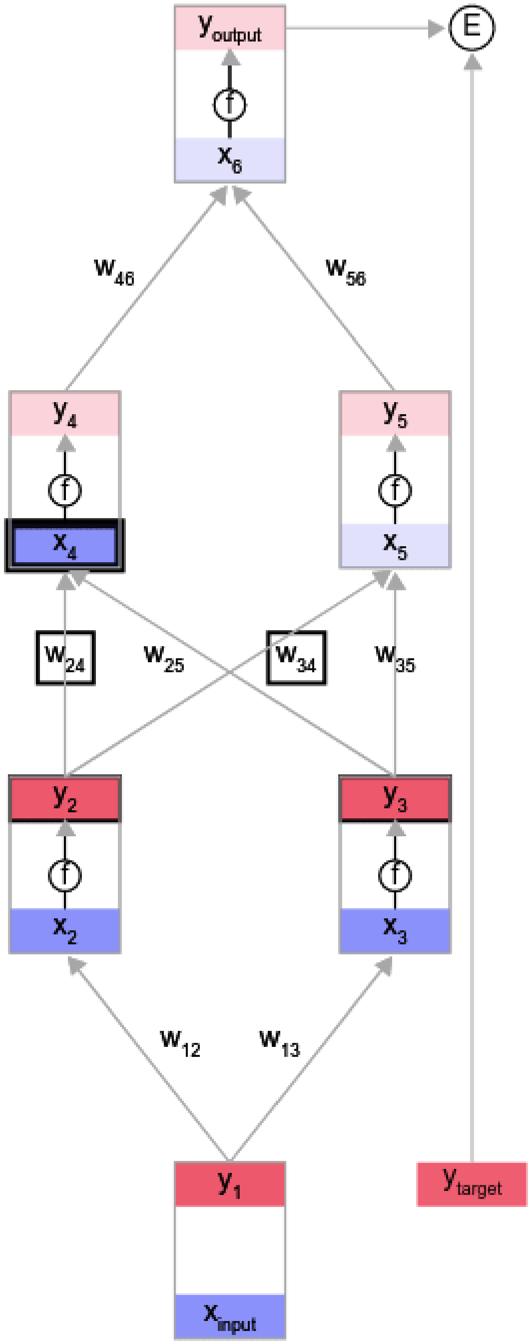


Forwardpropagation

Jedes Neuron im Netzwerk verbreitet die Eingabe im Rest des Netzwerks, um die Ausgabe zu berechnen:

$$y = tanh(x)$$

$$x_j = \sum_{i \in in(j)} w_{ij} y_i + b_j$$

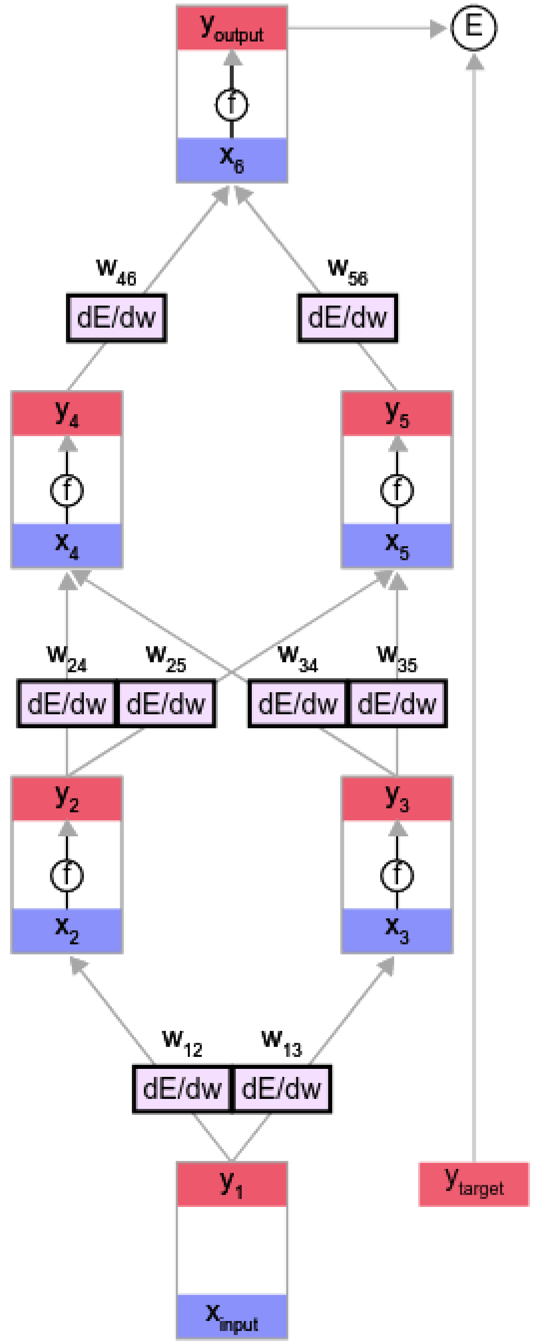


Gradientenabstieg

Der Backpropagation-Algorithmus berechnet die Aktualisierungsmenge der Gewichte basierend auf der Änderung der Fehlerfunktion in Bezug auf jedes Gewicht $\frac{dE}{dw_{ij}}$.

Die Gewichtsaktualisierung folgt dem **Gradientenabstieg**:

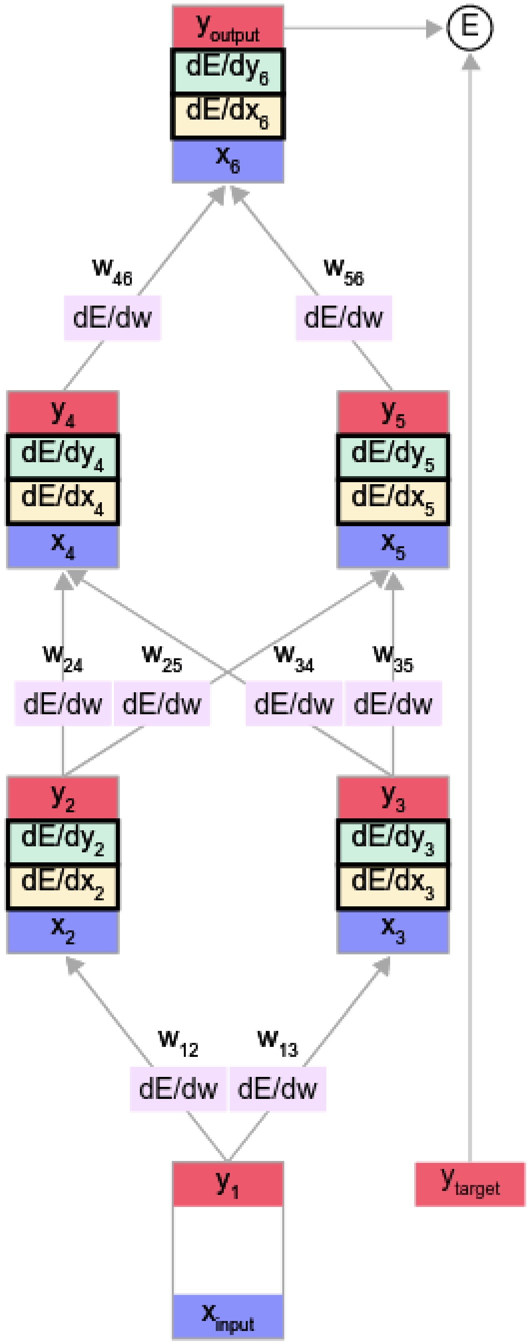
$$w_{ij} = w_{ij} - \alpha \frac{dE}{dw_{ij}}$$



Gradientenabstieg

Um $\frac{dE}{dw_{ij}}$ zu berechnen, müssen wir ermitteln, wie der Fehler ändert sich in Abhängigkeit von :

- dem Gesamteingang des Neurons $\frac{dE}{dx}$ und
- der Ausgabe des Neurons $\frac{dE}{dy}$.



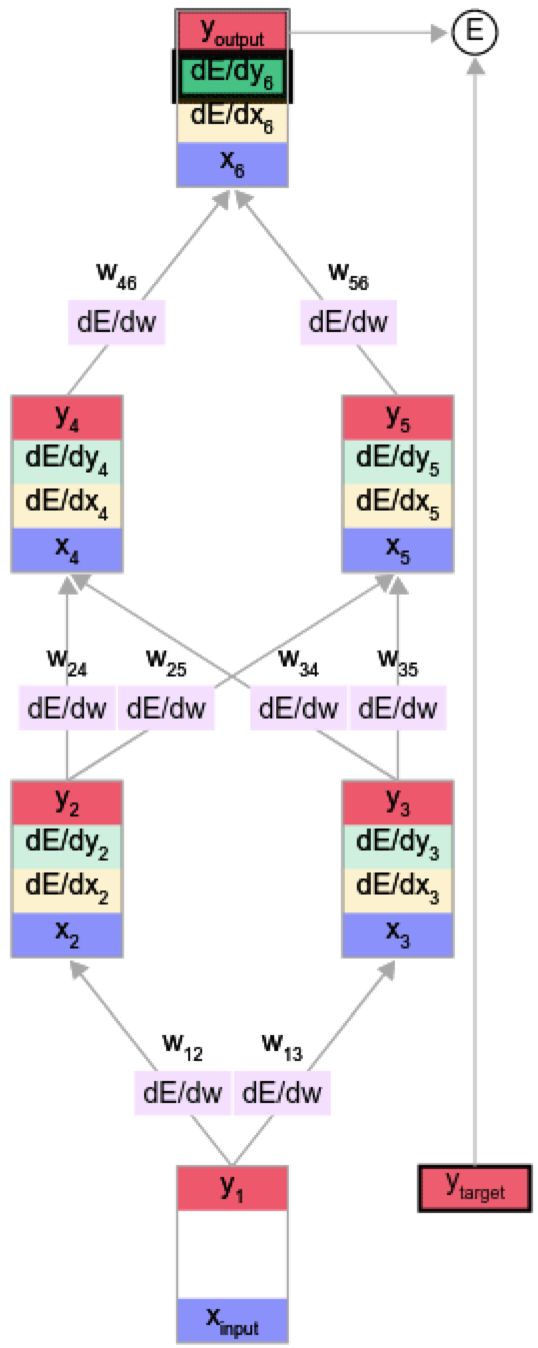
Backpropagation

Die Ableitung des Fehlers für unseren Klassifikator

$$E = \frac{1}{2}(y_{output} - y_{target})^2$$

ist

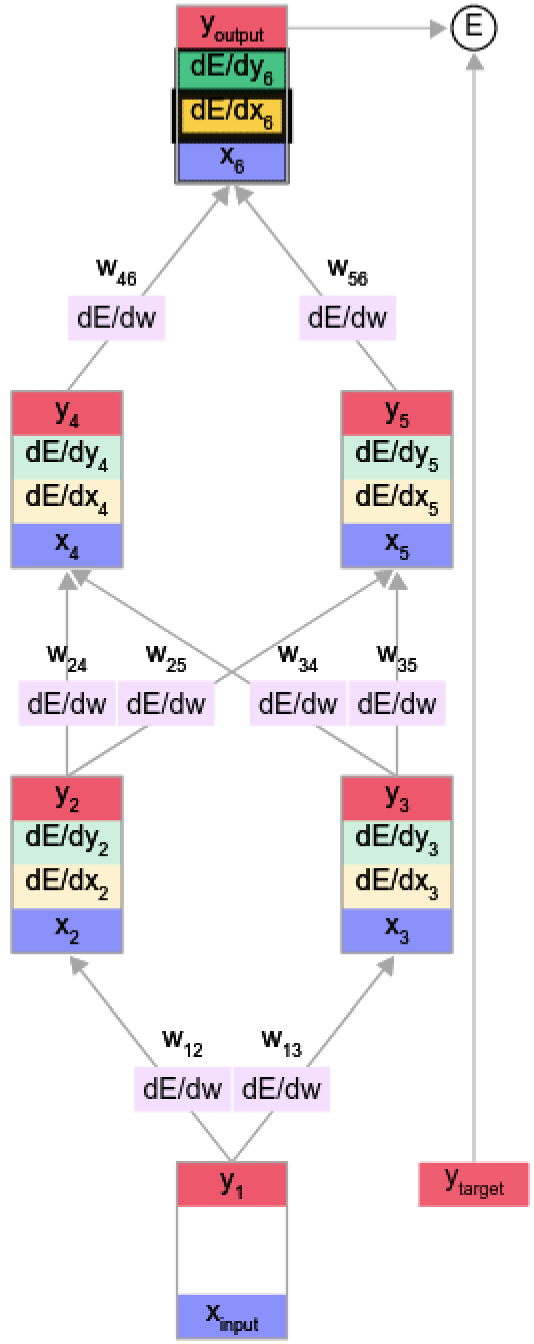
$$\frac{\partial E}{\partial y_{output}} = y_{output} - y_{target}$$



Backpropagation

Da nun $\frac{dE}{dy}$ verfügbar ist, kann $\frac{dE}{dx}$ mit der Kettenregel aus der vorigen Ebene errechnet werden:

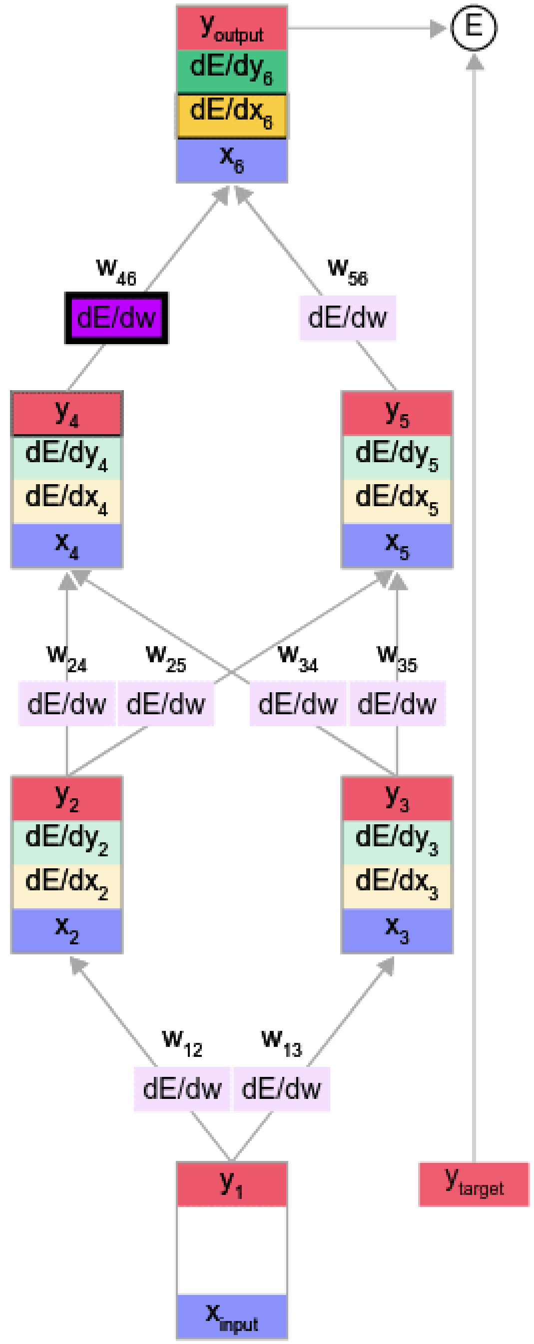
$$\frac{\partial E}{\partial x} = \frac{dy}{dx}\frac{\partial E}{\partial y} = \frac{d}{dx}f(x)\frac{\partial E}{\partial y}$$



Backpropagation

Sobald wir die Fehlerableitung bezüglich der Eingabe eines Neurons haben, können wir die Fehlerableitung bezüglich der in dieses Neuron führenden Gewichte erhalten:

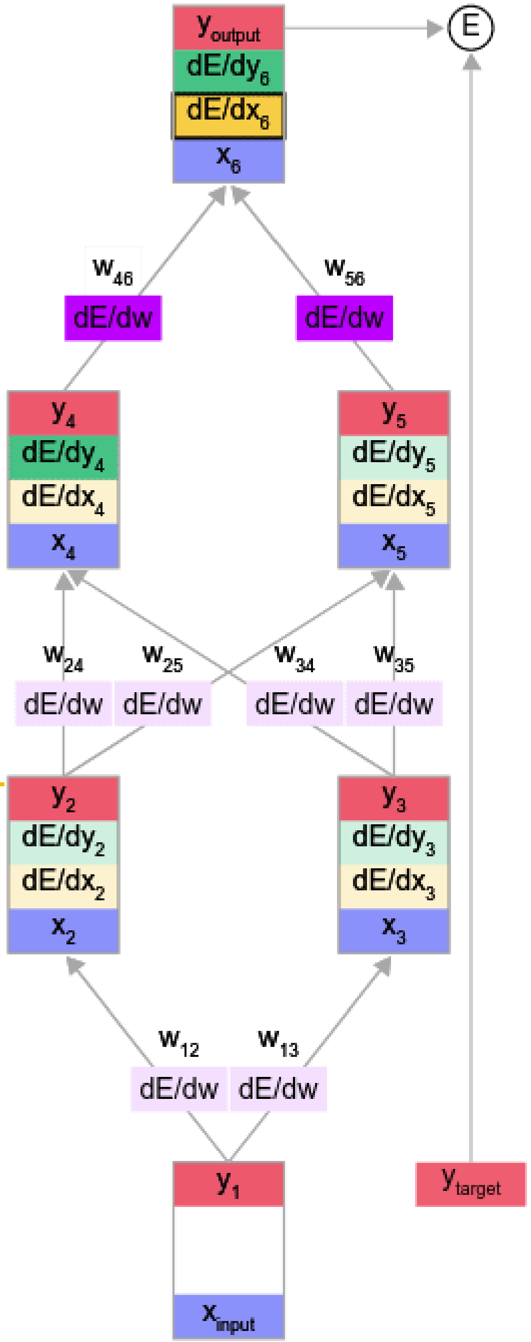
$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial x_j}{\partial w_{ij}} \frac{\partial E}{\partial x_j} = y_i \frac{\partial E}{\partial x_j}$$



Backpropagation

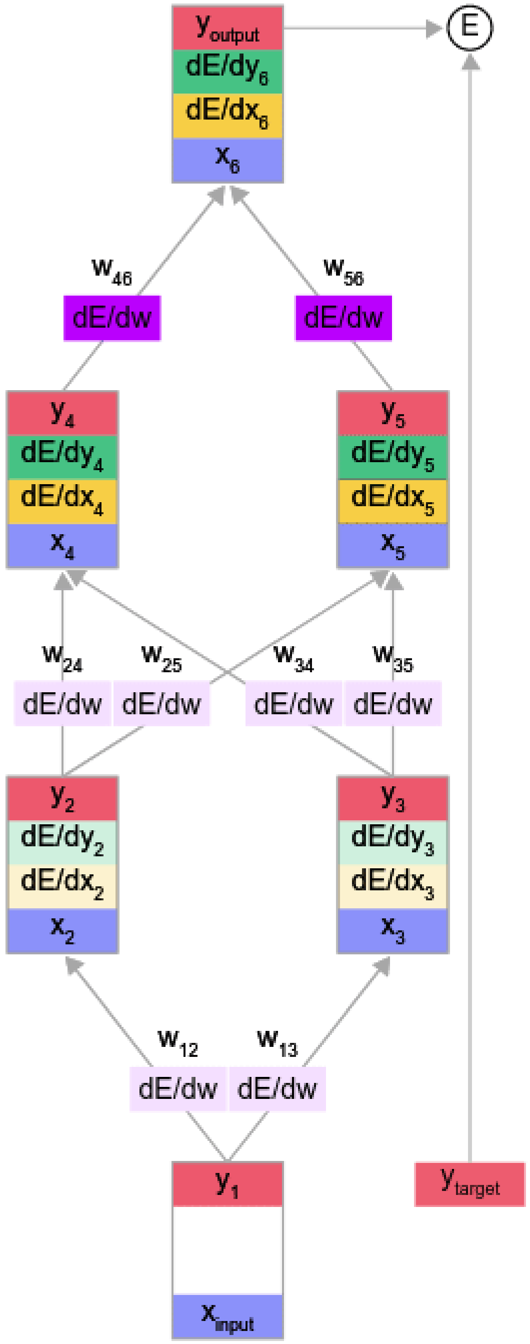
Ebenso kann $\frac{dE}{dy}$ mit der Kettenregel aus der vorigen Ebene errechnet werden:

$$\color{green}{\frac{\partial E}{\partial y_i}} = \sum_{j \in out(i)} \frac{\partial x_j}{\partial y_i} \color{orange}{\frac{\partial E}{\partial x_j}} \color{black}{= \sum_{j \in out(i)} w_{ij}} \color{orange}{\frac{\partial E}{\partial x_j}}$$



Backpropagation

Das Netzwerk wiederholt die Schritte für jedes Eingabebeispiel für eine bestimmte Anzahl von Epochen.



Implementierung

In [1]:

```python
# Einfaches neuronales Netz zur binären Klassifikation
import random
import math
import time as t
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

%matplotlib inline

def f(x):
    return math.tanh(x)


def df(x):
    try:
        return 1 / (math.cosh(x) ** 2)
    except OverflowError:
        return 0


class Network:


    def __init__(self, l_in, l_hid, l_out, rate=0.05):
        self.num_layer_in = l_in
        self.num_layer_hid = l_hid
        self.num_layer_out = l_out

        self.w1 = []
        self.w2 = []

        # init weights inputs -> hidden
        for i in range(self.num_layer_in * self.num_layer_hid ):
            self.w1.append(random.randrange(-10, 10) * 0.01)

        # init weights hidden -> output
```

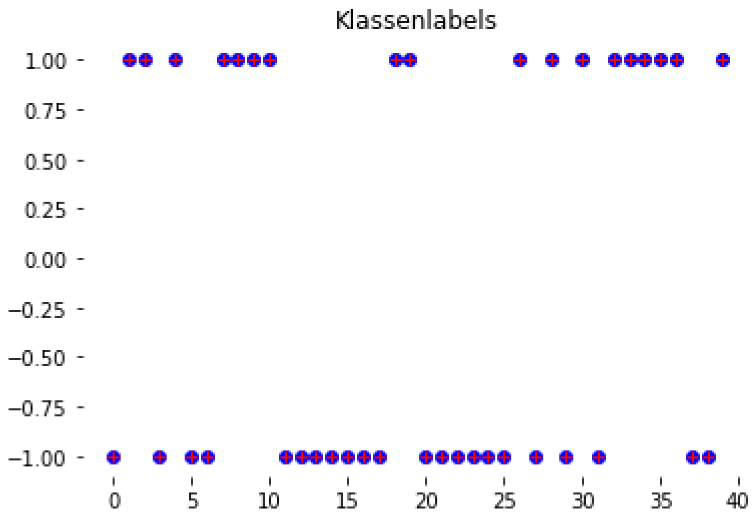
```
        for i in range(self.num_layer_hid * self.num_layer_out):
            self.w2.append(random.randrange(-10, 10) * 0.01)

        # init weights bias -> hidden
        for i in range(self.num_layer_hid):
            self.w1.append(random.randrange(-10, 10) * 0.01)

        # init weights bias -> output
        for i in range(self.num_layer_out):
            self.w2.append(random.randrange(-10, 10) * 0.01)

        self.learning_rate = rate

    def forward_propagation(self, data_in):
        net_hid = 0
        out_hid = []
        net_out = 0

        # Calculate net input and outputs for hidden layer
        for i in range(self.num_layer_hid):
            net_hid += float(data_in[0]) * self.w1[i]
            net_hid += float(data_in[1]) * self.w1[i + self.num_layer_hid]
            net_hid += self.w1[i + self.num_layer_in * self.num_layer_hid]
            out_hid.append(f(net_hid))

        # Calculate net input and output for output layer
        for i in range(self.num_layer_hid):
            net_out += out_hid[i] * self.w2[i]
        net_out += self.w2[self.num_layer_hid * self.num_layer_out]

        return f(net_out)

    def train(self, train_data):
        for data in train_data:
            net_hid = []
            out_hid = []
            net_out = 0
            out_out = 0
```

```
            updates_w1 = []
            updates_w2 = []

            # Calculate net input and outputs for hidden layer
            net = 0
            for i in range(self.num_layer_hid):
                net += float(data[0]) * self.w1[i]
                net += float(data[1]) * self.w1[i + self.num_layer_hid]
                net += self.w1[i + self.num_layer_in * self.num_layer_hid]
                net_hid.append(net)
                out_hid.append(f(net))

            # Calculate net input and output for output layer
            net = 0
            for i in range(self.num_layer_hid):
                net += out_hid[i] * self.w2[i]
            net += self.w2[self.num_layer_hid * self.num_layer_out]
            net_out = net
            out_out = f(net)

            # w2 weights update
            for i in range(len(self.w2)):
                if i < len(self.w2) - self.num_layer_out:
                    update = self.learning_rate * (float(data[2]) - out_out) * df(net_ou
t) * out_hid[i]
                else:
                    update = self.learning_rate * (float(data[2]) - out_out) * df(net_ou
t)
                updates_w2.append(update)

            # w1 weights update
            for i in range(len(self.w1)):
                if i < len(self.w1) - self.num_layer_hid:
                    if i < 4:
                        update = self.learning_rate * (float(data[2]) - out_out) * df(ne
t_out) * self.w2[i % 4] * df(net_hid[i % 4]) * float(data[0])
                    else:
```

```
                    update = self.learning_rate * (float(data[2]) - out_out) * df(ne
t_out) * self.w2[i % 4] * df(net_hid[i % 4]) * float(data[1])
                else:
                    update = self.learning_rate * (float(data[2]) - out_out) * df(net_ou
t) * self.w2[i % 4] * df(net_hid[i % 4])
                updates_w1.append(update)

            # update weights w1
            for i, update in enumerate(updates_w1):
                self.w1[i] += update

            # update weights w2
            for i, update in enumerate(updates_w2):
                self.w2[i] += update


def main():
    network = Network(2, 5, 1)
    train_data = []
    f = open('./input_dataset.in')

    try:

        while 1:
            new_in = f.readline().split(',')
            if len(new_in) < 3:
                break
            train_data.append(new_in)


        for _ in range(100):
            network.train(train_data)

        f = open('./testing_dataset.in')
        out_data = []
        test_data = []
        out_data_prob = []
        while 1:
```

```
            new_in = f.readline().split(',')
            if len(new_in) < 2:
                break
            new_in[0] = new_in[0][new_in[0].rfind(' ') + 1:]
            new_in[1] = new_in[1][new_in[1].rfind(' ') + 1:]
            output = network.forward_propagation(new_in)
            test_data.append(new_in)
            out_data_prob.append(output)
            if output > 0:
                output = '+1'
            else:
                output = '-1'
            out_data.append(output)

        f = open('./expected_dataset.in')
        out_expect_data = []
        while 1:
            new_in = f.readline().split('\n')
            if len(new_in) < 2:
                break
            out_expect_data.append(new_in[0])

    except EOFError:
        print('', end='')

    plt.figure(3)
    plt.plot(out_data, 'bo')
    plt.plot(out_expect_data,'r+')
    plt.title('Klassenlabels')
    plt.box("off")
    plt.savefig('output-klass.png')
    plt.show()

    # AUC / ROC curves
    from sklearn.metrics import roc_curve
    from sklearn.metrics import roc_auc_score
    from matplotlib import pyplot
    lr_auc = roc_auc_score(list(map(int, out_expect_data)), out_data_prob)
```

```
    # summarize scores
    print('NN AUC=%.3f' % (lr_auc))
    # calculate roc curves
    lr_fpr, lr_tpr, _ = roc_curve(list(map(int, out_expect_data)), out_data_prob)
    # plot the roc curve for the model
    pyplot.plot(lr_fpr, lr_tpr, marker='.')
    # axis labels
    pyplot.xlabel('False Positive Rate')
    pyplot.ylabel('True Positive Rate')
    pyplot.box("off")
    # show the legend
    pyplot.savefig('output-roc.png')
    # show the plot
    pyplot.show()

if __name__ == '__main__':
    main()
```



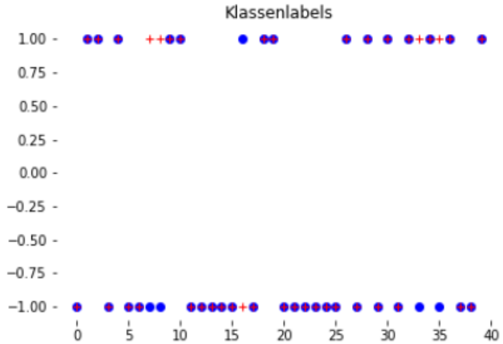

```
NN AUC=1.000
```

Analyse des Gradientenabstiegs

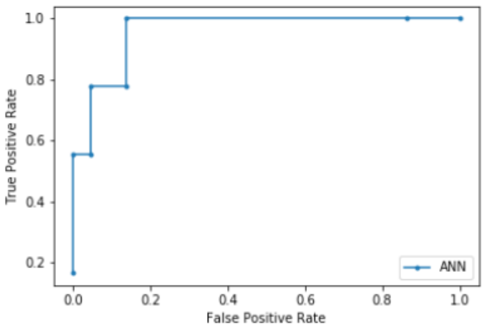
Abhängig von der Datenmenge machen wir einen **Kompromiss zwischen die Genauigkeit** und der **Konvergenzgeschwindigkeit** [2, 4].

Analyse des Gradientenabstiegs : Genauigkeit

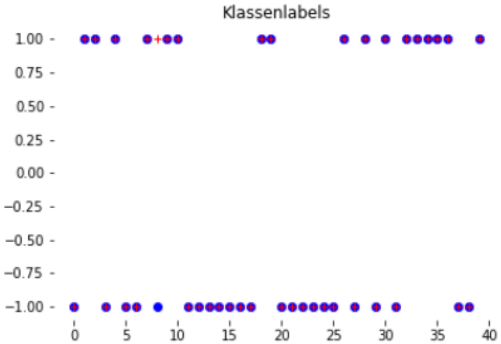
Erste Ausführung



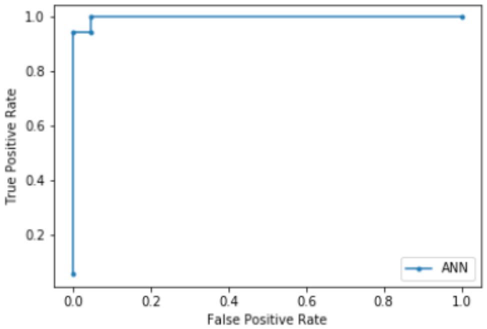
AUC ROC: 96%



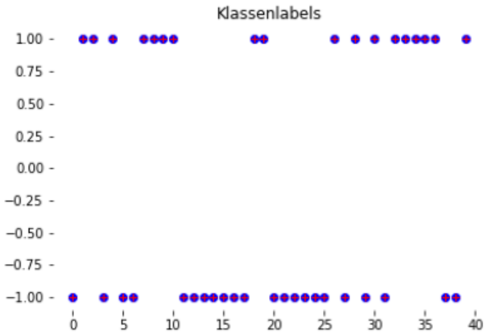
Zweite Ausführung



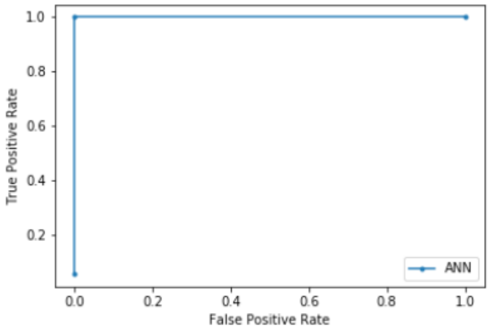
AUC ROC: 99,7%



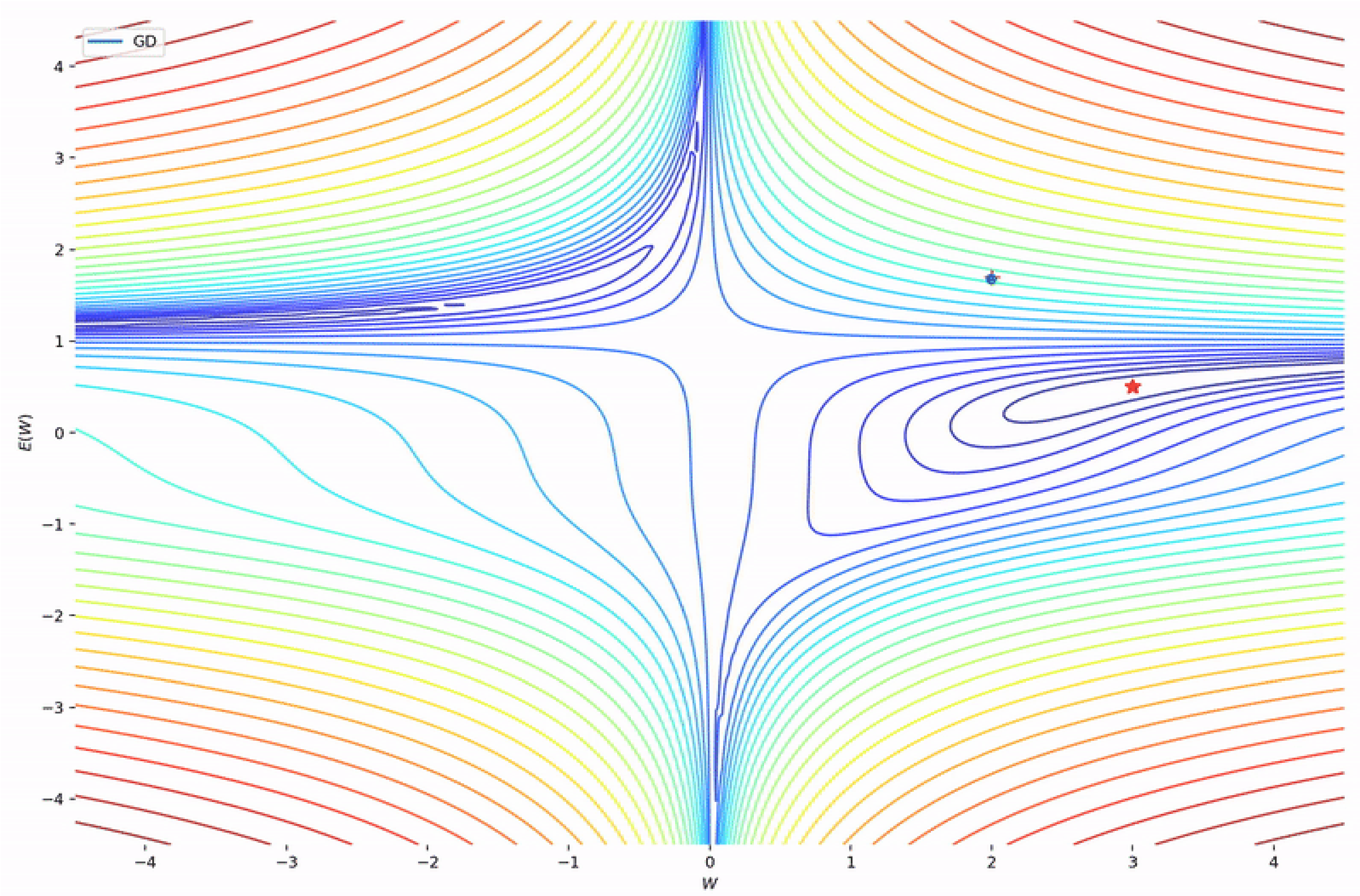
Dritte Ausführung



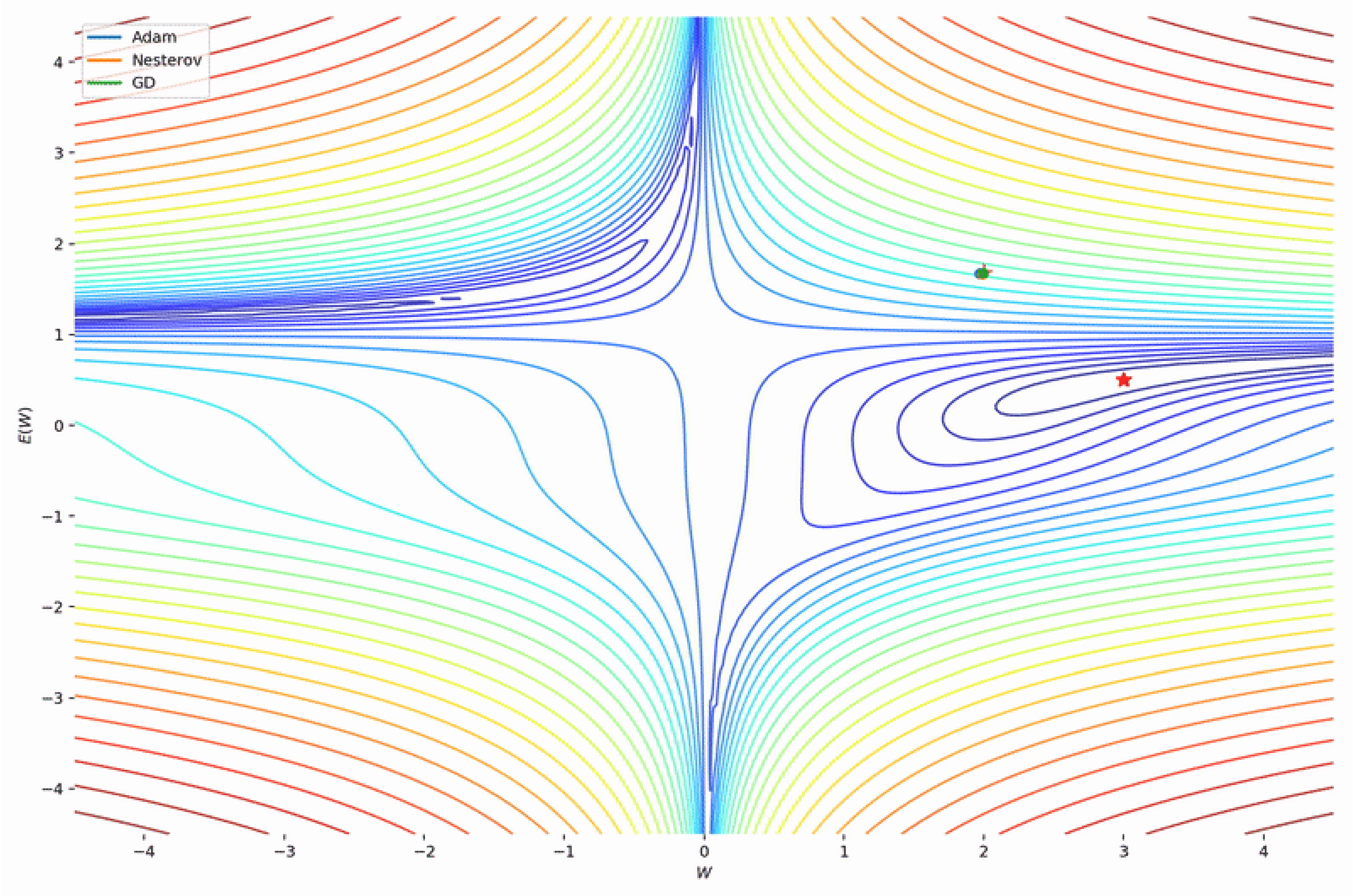
AUC ROC: 100%



Analyse des Gradientenabstiegs : Konvergenzgeschwindigkeit



Analyse des Gradientenabstiegs : Konvergenzgeschwindigkeit



Fazit

- **Gradientenabstieg** ist die **typische Optimierungsmethode** in neuronalen Netzen.
- **Lernen** in Neuronalen Netzen ist ein **iterativer Prozess**.
- Bei der **Backpropagation** wird ein Gradientenabstieg verwendet, um zu einer **Lösung zu konvergieren** (d. h. die **Gewichte zu finden**), die die **Fehlerfunktion minimiert**.
- **Gradientenabstieg** ist jedoch **problematisch** (Konvergenz, Präzision), aber es wurden viele **verbesserte Verfahren** entwickelt.
- Das **Verständnis des Gradientenabstieg** bei der Diagnose der Backpropagation **ist für erfolgreiche Anwendungen erforderlich**.

Literaturverzeichnis

[1] https://google-developers.appspot.com/machine-learning/ (https://google-developers.appspot.com/machine-learning/) (letzter Besuch, Dez 2019)

[2] Boyd, S., & Vandenberghe, L. (2004). Convex optimization. Cambridge university press.

[3] Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press.

[4] Ruder, S. (2016). An overview of gradient descent optimization algorithms. arXiv preprint arXiv:1609.04747.

Vorlesung Notebook herunterladen