Medizinische Bildanalyse
Wintersemester 2024/25

UNIVERSITÄT BONN

Kapitel 6a: Grundlagen Neuronaler Netze

Prof. Dr.-Ing. Thomas Schultz

URL: http://cg.cs.uni-bonn.de/schultz/
E-Mail: schultz@cs.uni-bonn.de
Büro: Friedrich-Hirzebruch-Allee 6, Raum 2.117

Per Video: 9. Januar 2025

b-it Bonn-Aachen International Center for Information Technology

Ehrungen für Pioniere des Tiefen Lernens

- **Turing-Preis 2018** an Yoshua Bengio, Geoffrey Hinton und Yann LeCun, für Durchbrüche zur Etablierung tiefer neuronaler Netze

- **Physik-Nobelpreis 2024** an John Hopfield und Geoffrey Hinton, für „grundlegende Entdeckungen und Erfindungen die maschinelles Lernen mit neuronalen Netzen ermöglichen“


Ill. Niklas Elmehed © Nobel Prize Outreach
John J. Hopfield


Ill. Niklas Elmehed © Nobel Prize Outreach
Geoffrey Hinton

https://www.acm.org/articles/bulletins/2019/march/turing-award-2018
https://www.nobelprize.org/prizes/physics/2024/summary/

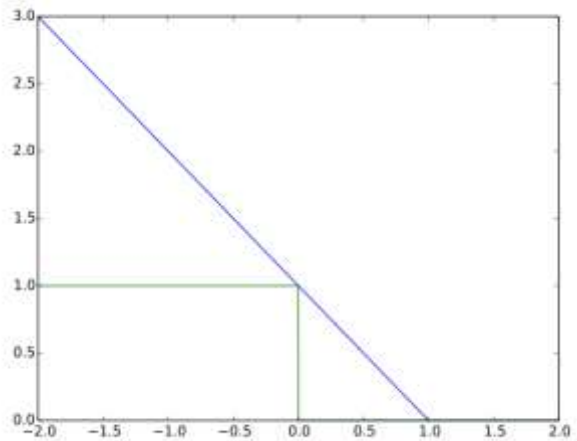
6a.1 Grundkonzepte des Maschinellen Lernens

Grundidee: Überwachtes Maschinelles Lernen

- Ziel des **überwachten maschinellen Lernens** ist es, aus **Trainingsbeispielen** $\{\mathbf{x}_i, y_i\}$ eine sinnvolle Funktion $y = f(\mathbf{x})$ zu lernen
  - **Klassifikation**: y ist diskret
    - Spezialfall: binär, z.B. $y = \pm 1$
    - *Beispiele*: Zeigt das Bild **x** Hautkrebs? Gehört Pixel **x** zu einer Zellmembran?
  - **Regression**: y ist kontinuierlich
    - *Beispiel*: Wie alt ist der Proband, von dem Hirnscan **x** stammt?
- *Grundannahme*: **Merkmalsvektoren** $\mathbf{x}_i$ und **Labels** y_i sind unabhängige Stichproben einer festen Wahrscheinlichkeitsverteilung $P(\mathbf{x}, y)$
  - **Tiefes Lernen** ermöglicht es hochdimensionale Daten (z.B. Bilder) direkt als Eingabe **x** zu verwenden, statt sie auf einen Merkmalsvektor zu reduzieren

Verlust und Risiko

- Eine **Verlustfunktion** $L(y, y')$ quantifiziert den Schaden der Vorhersage $y' = f(\mathbf{x})$, wenn y korrekt ist
  - *Binäre Klassifikation*:
    - Null-Eins-Verlust: Null wenn $y = y'$, sonst eins
    - Scharnier-Verlust (*engl*. hinge loss): *L*=max(0,1-*yy'*)
  - *Regression*:
    - Quadratischer Verlust (Gauß-Verlust): $L=(y-y')^2$

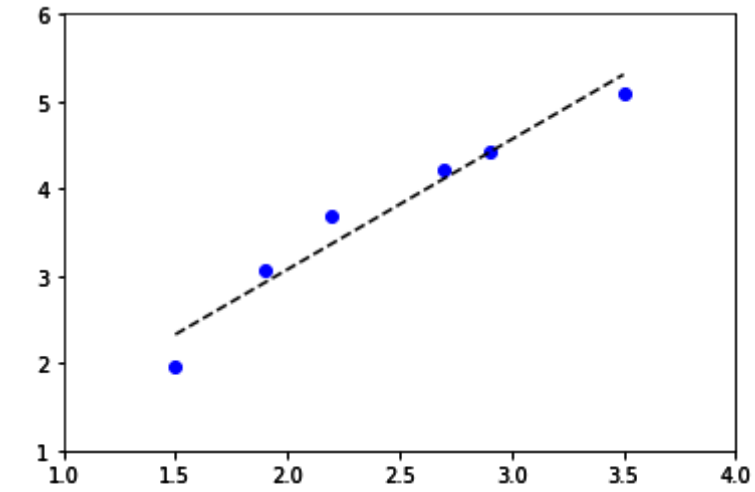


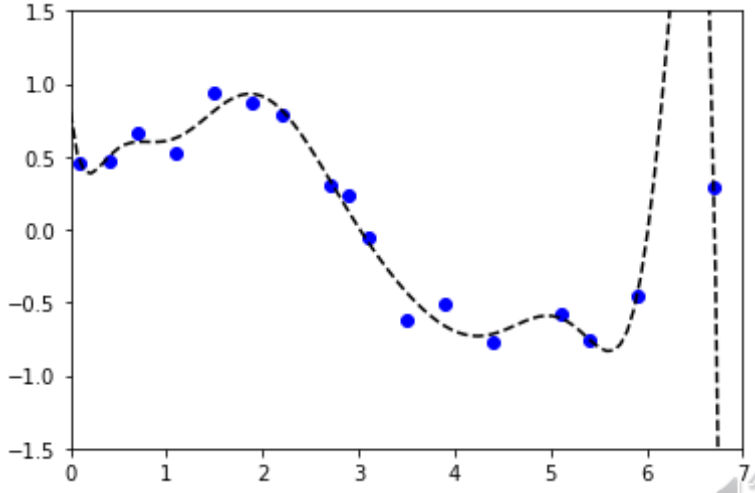
Hinge loss mit *y*=1

- Ziel des Trainings ist Minimierung des **Risikos,** d.h. des erwarteten Verlusts von $f(\mathbf{x})$ unter der Verteilung $P(\mathbf{x}, y)$
  - *Grundannahme*: $f(\mathbf{x})$ soll auf Daten angewandt werden, die aus derselben Verteilung stammen wie die Trainingsdaten

Training als Optimierungsproblem

- Die zu lernende Funktion $f(\mathbf{x})$ wird häufig durch **Parameter** bestimmt
  - *Beispiel:* Anpassung einer Ausgleichsgeraden an Messpunkte
    - Parameter: Achsenabschnitt, Steigung
- Das **Training** optimiert die Parameter im Hinblick auf eine Zielfunktion
  - Reine Fokussierung auf das **empirische Risiko,** d.h. den mittleren Verlust auf den Trainingsdaten $\{\mathbf{x}_i, y_i\}$, führt bei flexiblen Parametern leicht zu unplausiblen Funktionen
  - Regularisierung bevorzugt „einfache“ $f(\mathbf{x})$





Trainings- / Validierungs- / Testdaten

Bei der Entwicklung von Methoden mittels überwachten Lernens müssen die verfügbaren Daten partitioniert werden:

- Mittels der **Trainingsdaten** werden die Parameter optimiert
  - *Merke*: Geringer Trainingsfehler (empirisches Risiko) garantiert noch keine geeignete Generalisierung auf neue Daten!
- Mittels **Validierungsdaten** werden Hyperparameter eingestellt, z.B. Stärke der Regularisierung, Art der Optimierung
- Mittels **Testdaten** kann das Risiko von $f(\mathbf{x})$ geschätzt werden
  - *Wichtig*: Trainings-, Validierungs- und Testdaten müssen disjunkt sein. Vorsicht auch bei wiederholten Messungen desselben Patienten!
  - Bestenfalls werden Testdaten erst nach der Entwicklung verfügbar

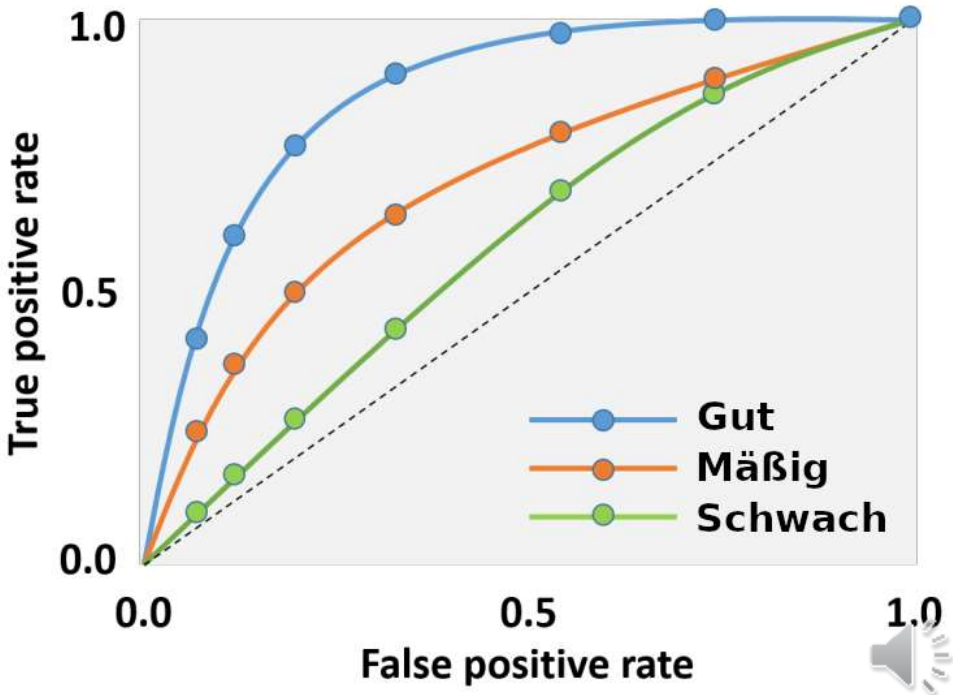
Evaluierung von Klassifikatoren

- Für binäre Klassifikatoren ergeben sich in der **Konfusionsmatrix** folgende Fälle
- Daraus leitet man ab:
  - **Korrektklassifikationsrate** (engl. accuracy) $ACC = \frac{\mathrm{RP}+\mathrm{RN}}{\mathrm{RP}+\mathrm{RN}+\mathrm{FP}+\mathrm{FN}}$
  - **Positiver Vorhersagewert** (engl. precision) $P = \frac{\mathrm{RP}}{\mathrm{RP}+\mathrm{FP}}$
  - **Sensitivität / Trefferquote** (engl. recall) $R = \frac{\mathrm{RP}}{\mathrm{RP}+\mathrm{FN}}$
  - **F-Maß** $F = 2\frac{P \cdot R}{P+R}$

		Vorhersage $f(\mathbf{x})$	
		Positiv	Negativ
Label y	Positiv	Richtig Positiv (RP)	Falsch Negativ (FN)
	Negativ	Falsch Positiv (FP)	Richtig Negativ (RN)

Evaluierung mittels ROC-Kurven

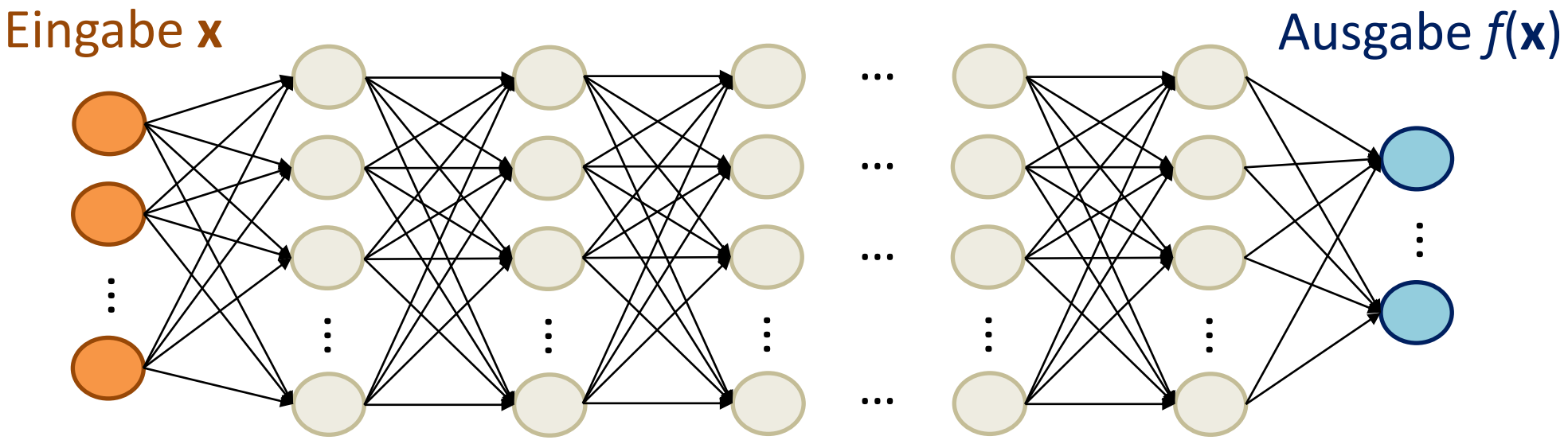
- Viele binäre Klassifikatoren basieren auf einer kontinuierlichen Entscheidungsfunktion. Durch Wahl verschiedener Schwellenwerte kann man die Balance einstellen zwischen
  - Sensitivität = **Richtig-Positiv-Rate** $\mathrm{TPR} = R = \frac{\mathrm{RP}}{\mathrm{RP+FN}}$
  - **Falsch-Positiv-Rate** $\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{RN+FP}}$
- Die **ROC-Kurve** (*engl*. receiver operating characteristic) zeigt die Auswirkungen verschiedener Einstellungen
  - ROC-Kurven nahe der Diagonalen entsprechen zufälliger Klassifikation
  - Die **Fläche unter der ROC-Kurve** (*engl.* Area under the curve, AUC) ist ein weiteres Maß für die Güte eines Klassifikators



6a.2 Grundlagen Neuronaler Netze

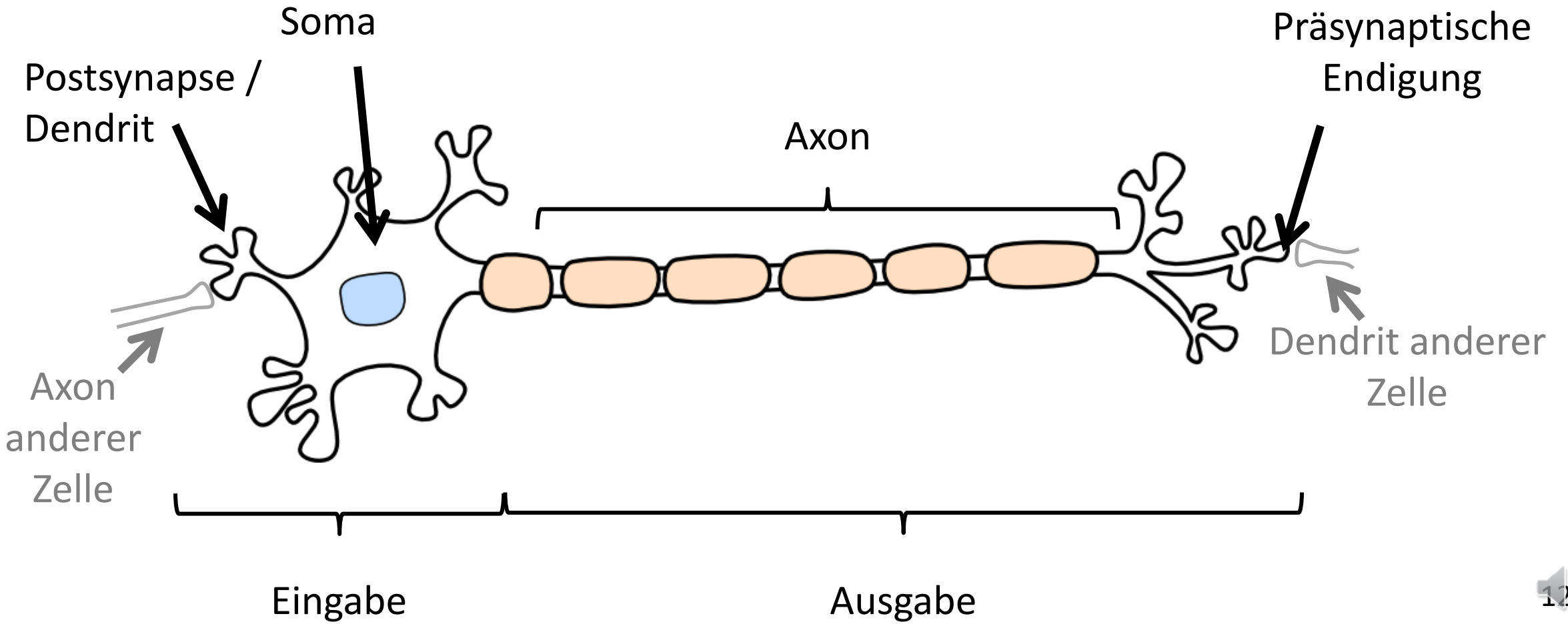
Grundidee Neuronaler Netze

- **Neuronale Netze** setzen die zu lernende Funktion $f(\mathbf{x})$ aus vielen einfachen Bausteinen zusammen, den künstlichen Neuronen
- Im **vorwärtsgerichteten** Fall (*engl*. feed forward) lassen sich die Neuronen so in Schichten anordnen, dass die Ausgaben jeder Schicht nur von späteren Schichten verarbeitet werden



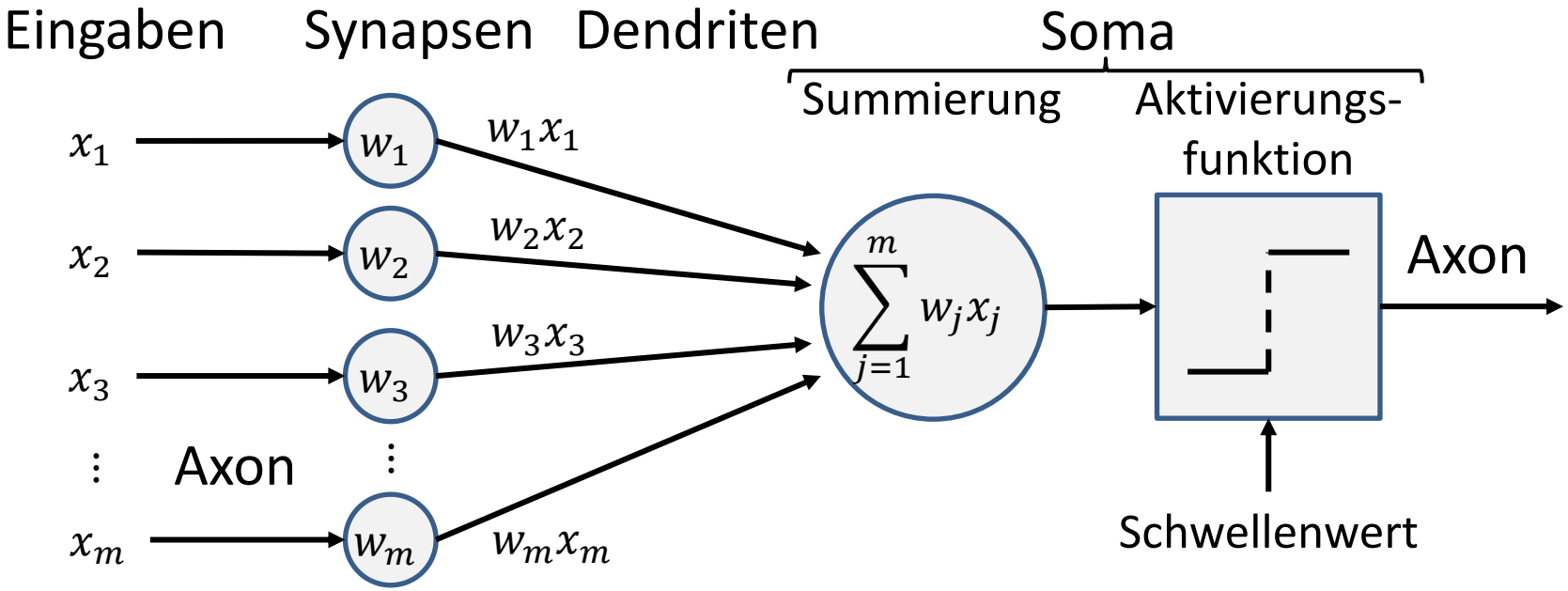
Inspiration: Das Biologische Neuron

- Neuronale Netze sind von der Informationsverarbeitung im menschlichen Gehirn inspiriert



Vom Biologischen zum Künstlichen Neuron

Biologisches Neuron	Künstliches Neuron
Axon	Verbindung zwischen den Schichten
Synapse	Gewichte
Dendrit	Gewichtete Eingaben
Soma	Summierung und Aktivierungsfunktion



Künstliche Neurone: Formel

Die Ausgabe y_k des *k*-ten künstlichen Neurons einer Schicht ist durch folgende Formel gegeben:

$$y_k = f\left(\sum_{j=1}^{m} w_{kj} x_j + b_k\right)$$

- Eingaben x_j $(j = 1, \ldots, m)$
- Gewichte w_{kj}
- Negativer Schwellenwert (*engl*. bias) b_k
- Aktivierungsfunktion f

Hinweis: Diese Formel ist von biologischen Neuronen inspiriert, aber kein realistisches Modell!

Künstliche Neurone: Matrix-Notation

Häufig werden die Parameter aller Neurone einer Schicht in einer Matrix dargestellt. Hierzu führen wir ein immer-an-Neuron ein $(x_0 := 1, w_{k0} := b_k)$, so dass

$$y_k = f\left(\sum_{j=0}^{m} w_{kj} x_j\right)$$

Dies ermöglicht die Schreibweise

$$\mathbf{y} = f(\mathbf{W}\mathbf{x})$$

wobei f komponentenweise angewandt wird

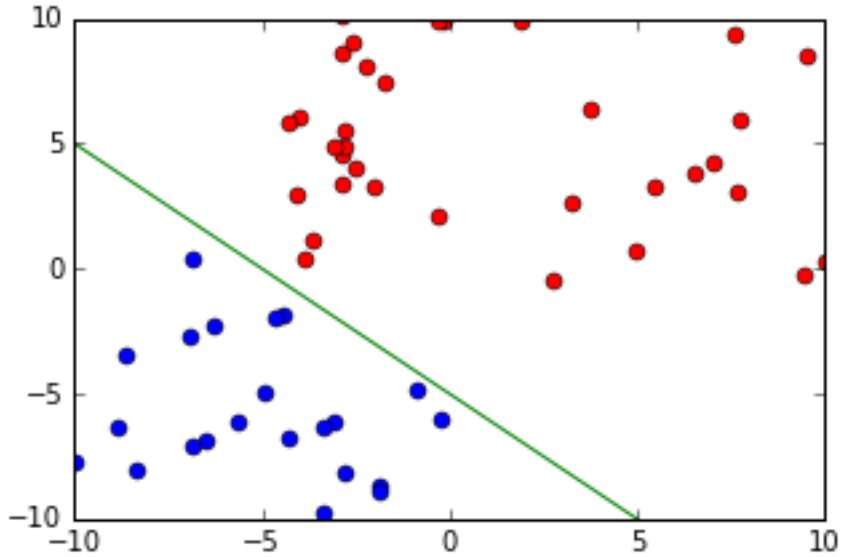
Quiz: An was erinnert Sie dieser Trick?

Aktivierungsfunktionen

- Biologische Neurone erzeugen ein **Aktionspotential**, wenn die gewichtete Summe der Eingaben eine Schwelle überschreitet
- Definieren wir analog $\alpha_k := \sum_{j=1}^{m} w_{kj} x_j + b_k$ und eine binäre **Aktivierungsfunktion**

$$f(\alpha_k) = \begin{cases} 1 & \text{wenn } \alpha_k > 0 \\ 0 & \text{wenn } \alpha_k \leq 0 \end{cases}$$

erhalten wir einen
linearen Klassifikator



Beispiel: Lineare Bildklassifikation

Eine **Bildklassifikation** mit einer einzigen Schicht von Neuronen entspräche einer affinen Funktion, die jeder Klasse einen Wert zuweist, der mit der Wahrscheinlichkeit der Klasse zunehmen soll

$$f(x,W,b)=Wx+b=s$$

x: Eingabe (Bild)
W: Parameter (Gewichte)
b: Bias (Versatz)
s: Bewertung (*engl*. Score) jeder Klasse

x

f(x,W,b)

-2.5	Katze
2	Hund
3	Vogel

Training einer Linearen Bildklassifikation

Ziel des **Trainings** einer linearen Bildklassifikation wäre es, die Parameter **W** und **b** so zu wählen, dass bei möglichst vielen Trainingsbildern $\mathbf{x}_i$ die korrekte Klasse den höchsten Wert hat

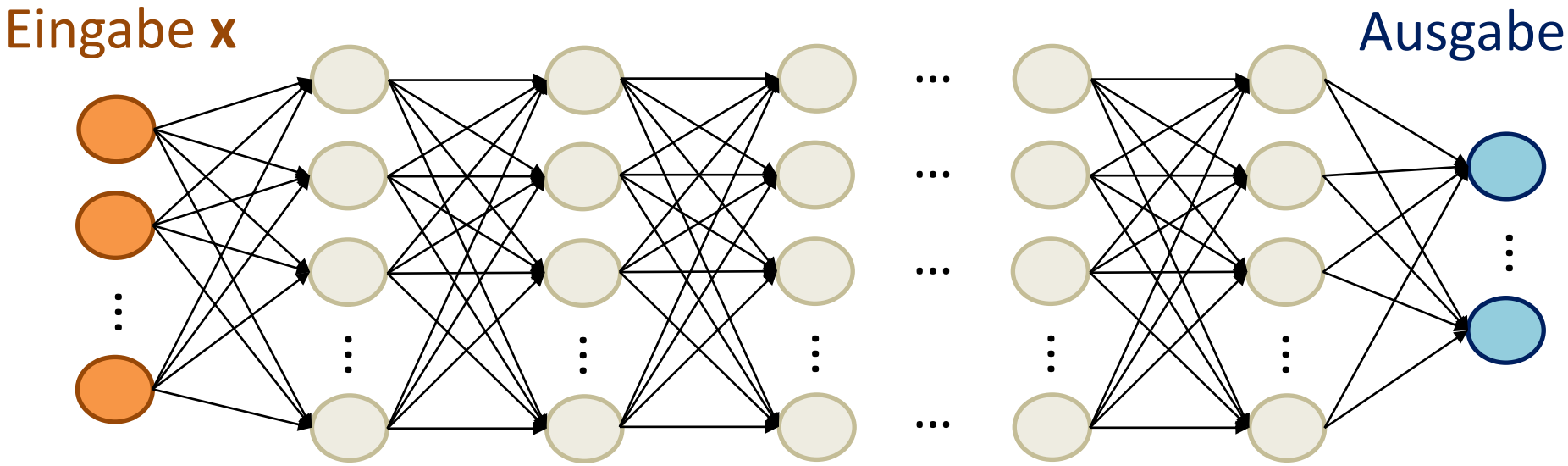
- Hierzu werden wir eine Verlustfunktion nach **W** und **b** ableiten

64x64
RGB-Bild

W				x_i	b	$f(x_i,W,b)=Wx_i+b$	
2.3	5	0	...	3	-1.5	-2.5	Katze
-1	-3	1	...	231	2	2	Hund
0.5	2	-1	...	21	1	1	Vogel
				⋮			

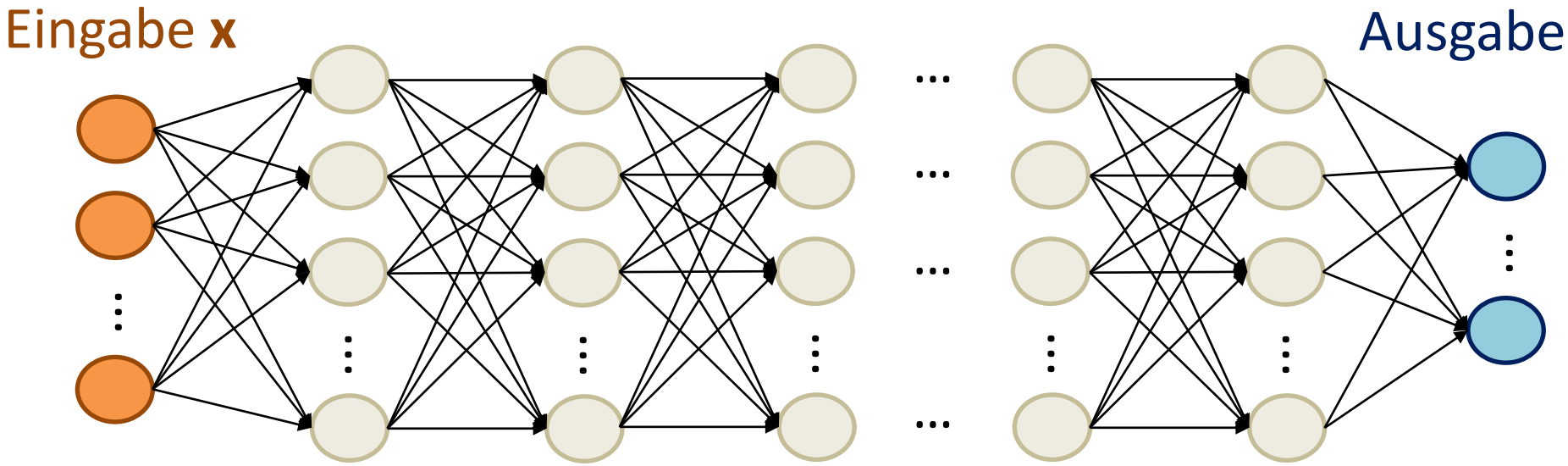
Tiefe Neuronale Netze

- Komplexe Aufgaben wie Bildklassifikation erfordern komplizierte nichtlineare Funktionen $f(\mathbf{x})$
- **Tiefe neuronale Netze** ermöglichen dies durch die Anordnung künstlicher Neurone in einer hohen Zahl von Schichten
  - Für die Bildklassifikation können das mehr als 100 sein
  - „Tiefe“ Netze haben mindestens zwei verborgene Schichten



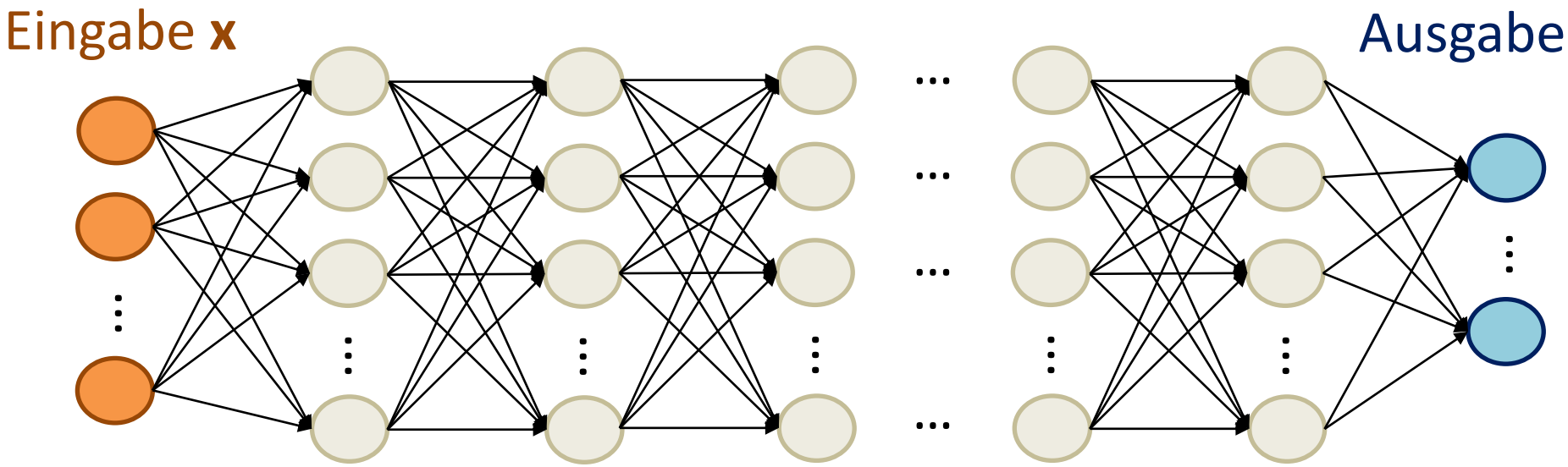
Verborgene Schichten

- Ein- und Ausgaben sind aus den Trainingsdaten bekannt
  - Da die Eingabe keine Gewichte hat, zählt sie nicht als eigene Schicht
- Schichten zwischen Ein- und Ausgabe werden als „**verborgen**“ bezeichnet (*engl*. hidden layers)
  - Ermöglichen hierarchischen Aufbau abstrakter Repräsentationen
  - Sind für Menschen häufig nicht klar interpretierbar



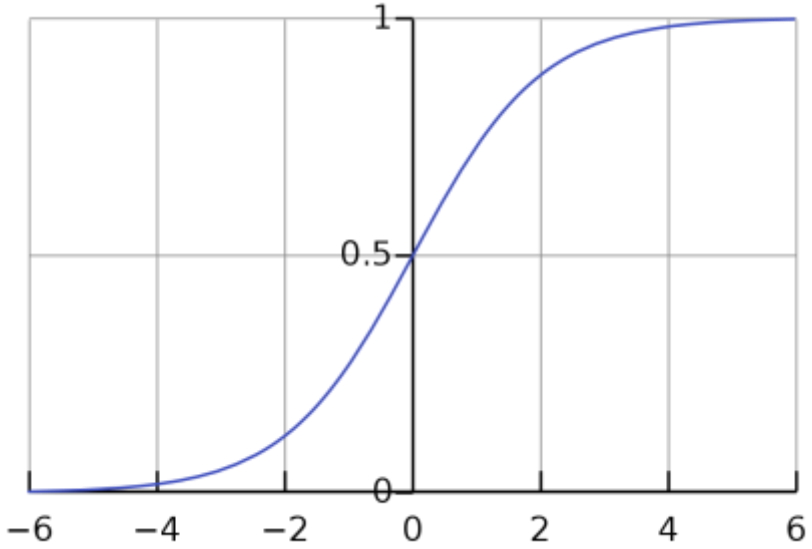
Vollständig verbundene Schichten

- Zwei Schichten sind **vollständig verbunden**, wenn jedes Neuron der ersten mit jedem der zweiten verbunden ist
- Eine einzige Schicht ergibt einen linearen Klassifikator
  - *Quiz*: Warum bieten tiefere Netze uns nur dann einen Vorteil, wenn wir eine nichtlineare Aktivierungsfunktion $f(\mathbf{W}\mathbf{x})$ nutzen?



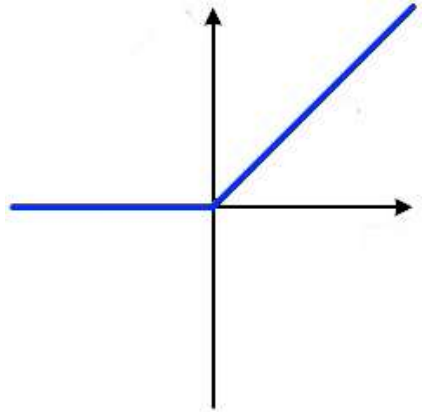
Sigmoide Aktivierungsfunktion

- **Logistische Funktion** $f(\alpha) = \frac{1}{1+e^{-\alpha}}$
  - Ausgabe in (0,1) als Wahrscheinlichkeit interpretierbar
  - Ähnliches Verhalten wie Sprungfunktion, aber sinnvoll differenzierbar
    - Voraussetzung für Training, siehe 6a.3
  - Ableitung einfach zu berechnen: $f'(\alpha) = f(\alpha)\big(1 - f(\alpha)\big)$
  - *Nachteil*: Ableitung für große/kleine α nahe Null
  - Wird manchmal für Ausgabe-Schichten verwendet, selten für innere Schichten



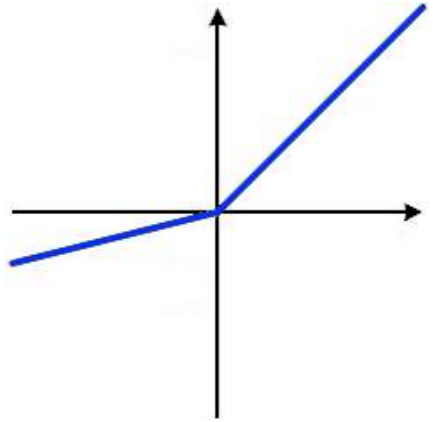
Rectified Linear Unit (ReLU)

- **Rectified Linear Unit (ReLU)** $f(\alpha) = \max(\alpha, 0)$
  - Lineare Einheit mit Rectifier (“Gleichrichter”)
  - *Biologische Motivation*: Neurone feuern unterhalb der Schwelle nicht, bei darüber hinaus zunehmender Erregung mit höherer Frequenz
  - $f(\alpha)$ und $f'(\alpha)$ einfach und effizient zu berechnen
  - Verringert Probleme mit verschwindenden Gradienten
  - Sehr beliebt in tiefen Netzwerken
  - *Nachteil*: ReLUs können „sterben“ – wenn keine Eingabe sie aktiviert, gibt es keinen Gradienten um die Gewichte zu ändern

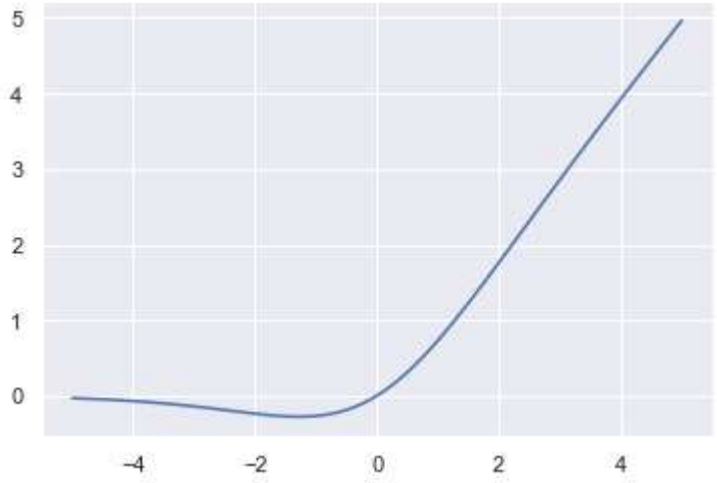


Varianten der ReLU

- **Leaky ReLU**[1] $f(\alpha) = \max(0.01\alpha, \alpha)$
  - *Ziel*: “Sterben” der ReLUs vermeiden
  - Parametrischer ReLU[2]: Statt 0.01 festzulegen wird dieser Faktor ebenfalls gelernt



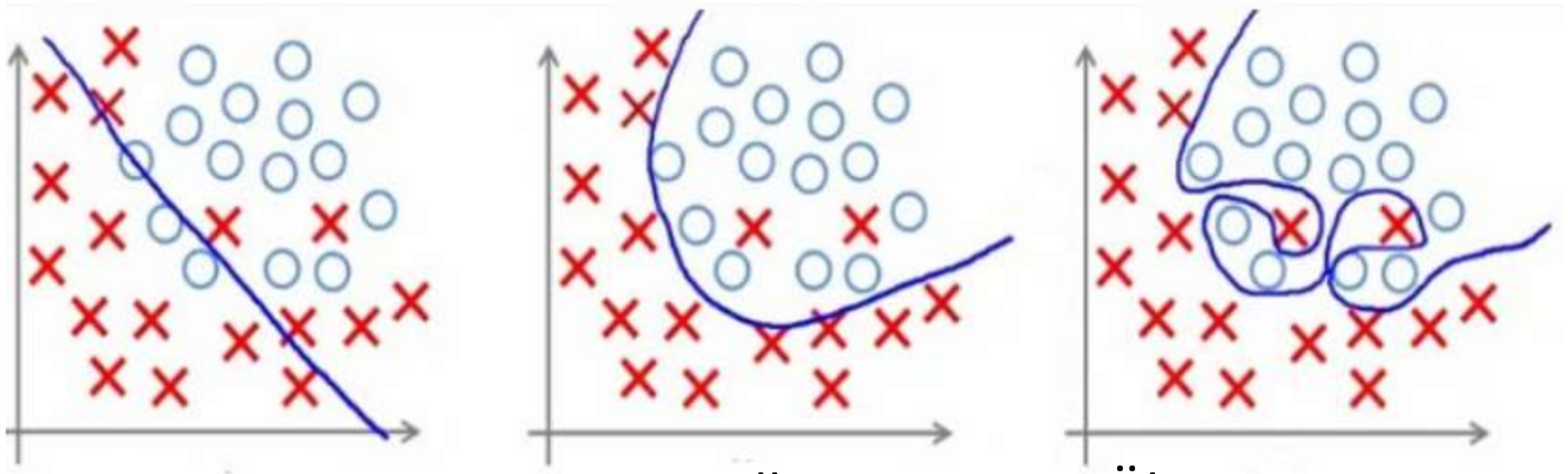
- **Swish**[3] $f(\alpha) = f(\alpha) = \alpha/(1 + e^{-\alpha})$
  - Überall differenzierbare Variante des ReLU
  - Nicht monoton
  - Führt bei tiefen Netzen häufig zu etwas besseren Ergebnissen
  - Erzeugt jedoch höheren Rechenaufwand



[1]Maas et al. ICML 2013 [2]He et al. ICCV 2015 [3]Ramachandran et al. ICLR Workshop 2018

Kapazität eines Neuronalen Netzwerks

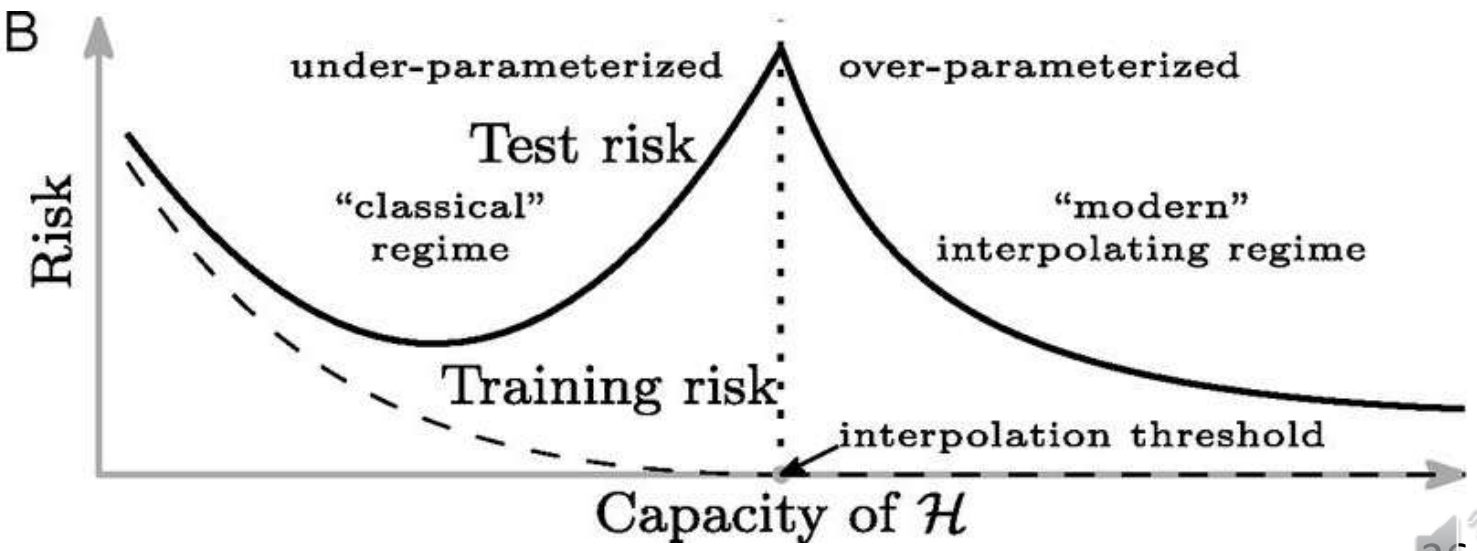
- Mit zunehmender Zahl von Neuronen pro Schicht und zunehmender Tiefe stellen neuronale Netzwerke immer flexiblere Familien von Funktionen $f(\mathbf{x})$ dar
- Zu hohe bzw. geringe Kapazität kann zu **Überanpassung** (*engl*. overfitting) bzw. **Unteranpassung** (*engl*. underfitting) führen



Unteranpassung Sinnvoller Fit Überanpassung

Phänomen des „Doppelten Abstiegs“

- In der Praxis erreicht man die besten Ergebnisse häufig mit neuronalen Netzen, deren Kapazität *höher* ist, als es zur Interpolation der Trainingsdaten nötig wäre
- *Erklärungsansatz*: Solche Netze können die Trainingsdaten auf unterschiedliche Art interpolieren – auch mit solchen Funktionen $f(\mathbf{x})$, die sinnvoll auf neue Daten generalisieren
  - Algorithmen zum Training der Netze (siehe 6a.3) scheinen „gute“ Lösungen zu bevorzugen
  - Es ist nicht völlig geklärt, warum das so ist



Bildquelle: Belkin et al., PNAS 2019

Zusammenfassung: Neuronale Netze

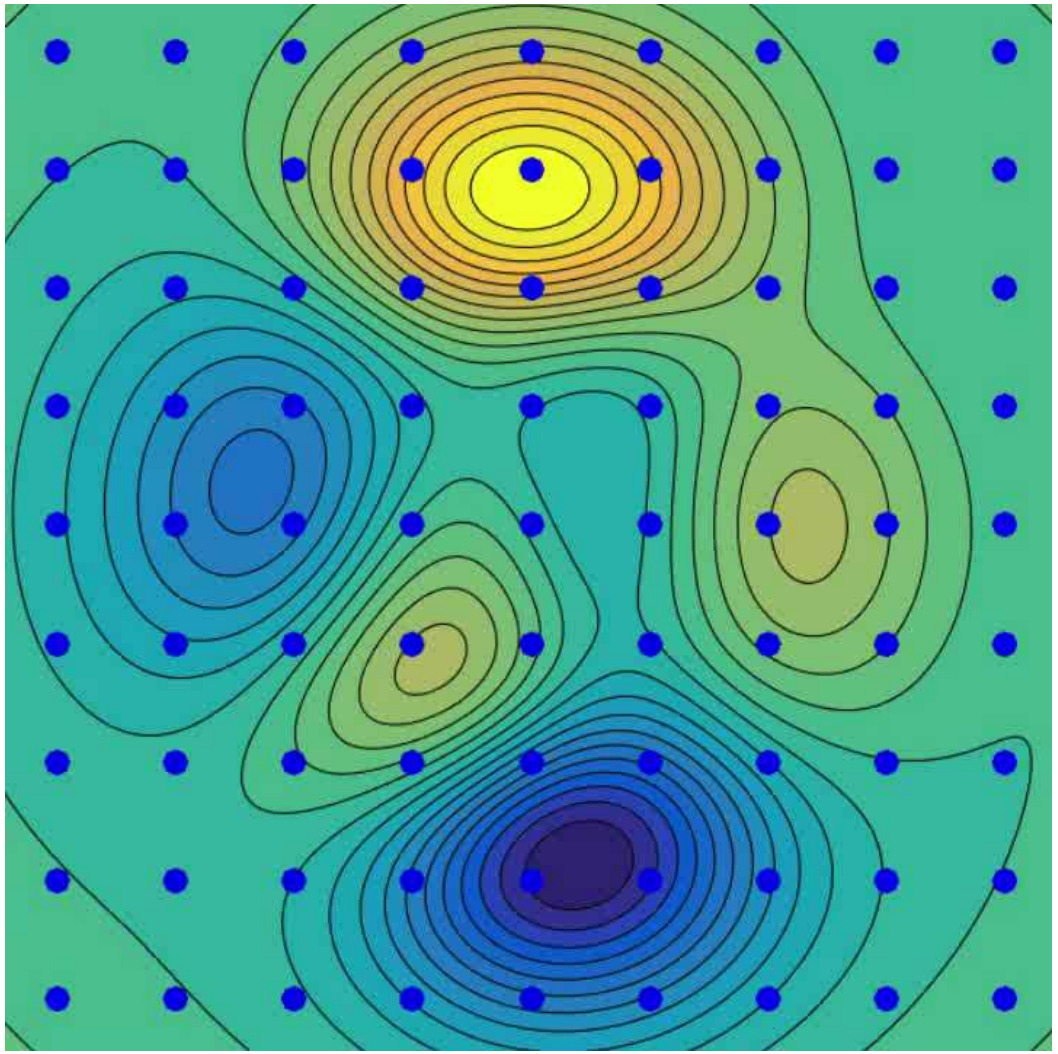
- **Neuronale Netze** sind eine beliebte Repräsentation lernbarer Funktionen $f(\mathbf{x})$ hochdimensionaler Eingaben
- **Künstliche Neurone** sind ihre Bausteine
  - Affine Abbildung mit nichtlinearer Aktivierungsfunktion $\mathbf{y} = f(\mathbf{W}\mathbf{x})$
  - Neuronale Netze mit **einer einzigen Schicht** ergeben lineare Klassifikatoren
  - **Tiefe neuronale Netze** ermöglichen nichtlineare Funktionen
    - Benötigen geeignete **Aktivierungsfunktionen** wie z.B. ReLU
- Die **Kapazität** eines Netzwerks wird durch die Zahl der Schichten und Neurone bestimmt
  - Die genauen Werte sind **Hyperparameter,** die z.B. mittels Validierungsdaten eingestellt werden können

6a.3 Training Neuronaler Netze

Grundidee: Training Neuronaler Netze

- Aufgabe des **Trainings** ist es, die Parameter des neuronalen Netzes (insb. Gewichte und Bias-Terme) so einzustellen, dass die Funktion $f(\mathbf{x})$ die gewünschte Aufgabe erfüllt
  - z.B. Bild $\mathbf{x}$ einer Hautveränderung als gutartig oder bösartig einstufen
- Wie in 6a.1 erfolgt dies durch Minimierung einer **Verlustfunktion** L, die auf den Trainingsdaten Abweichungen zwischen Ausgaben $f(\mathbf{x}_i)$ und Labeln y_i bestraft
  - *Grundlegende Strategie:* Gradientenabstieg, d.h. Berechnung der Ableitungen von L nach den Parametern und Veränderung der Parameter entgegen dieser Richtung
    - *Quiz*: In welchem Kontext haben wir bereits dieselbe Grundidee genutzt?

Illustration: Gradientenabstieg



Video: Gabriel Peyré via Wikimedia

Beispiel: Hinge-Loss für mehrere Klassen

- Der binäre Hinge-Loss aus 6a.1 lässt sich wie folgt auf mehrere Klassen verallgemeinern:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

  - y_i ist das korrekte Label für Trainingsbeispiel *i*
  - s_j ist der Wert für Klasse *j*
- Für alle *N* Trainingsdatenpunkte ergibt sich die Verlustfunktion

$$L = \frac{1}{N} \sum_i L_i$$



Katze	3.0	**3.0**	1.0
Hund	**1.0**	-5.0	3.2
Vogel	-2.5	1.5	**-5.0**
Verlust	3.0	0	16.2

Probabilistische Vorhersagen mittels Softmax

- **Softmax** rechnet die Bewertungen der Klassen in bedingte Wahrscheinlichkeiten um:

$$P(y = k|\mathbf{x} = \mathbf{x}_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

  - *Interpretation*: vom neuronalen Netz geschätzte Wahrscheinlichkeit, dass *k* das passende Label für $\mathbf{x}_i$ ist

Katze	3.0	**3.0**	1.0
Hund	**1.0**	-5.0	3.2
Vogel	-2.5	1.5	**-5.0**

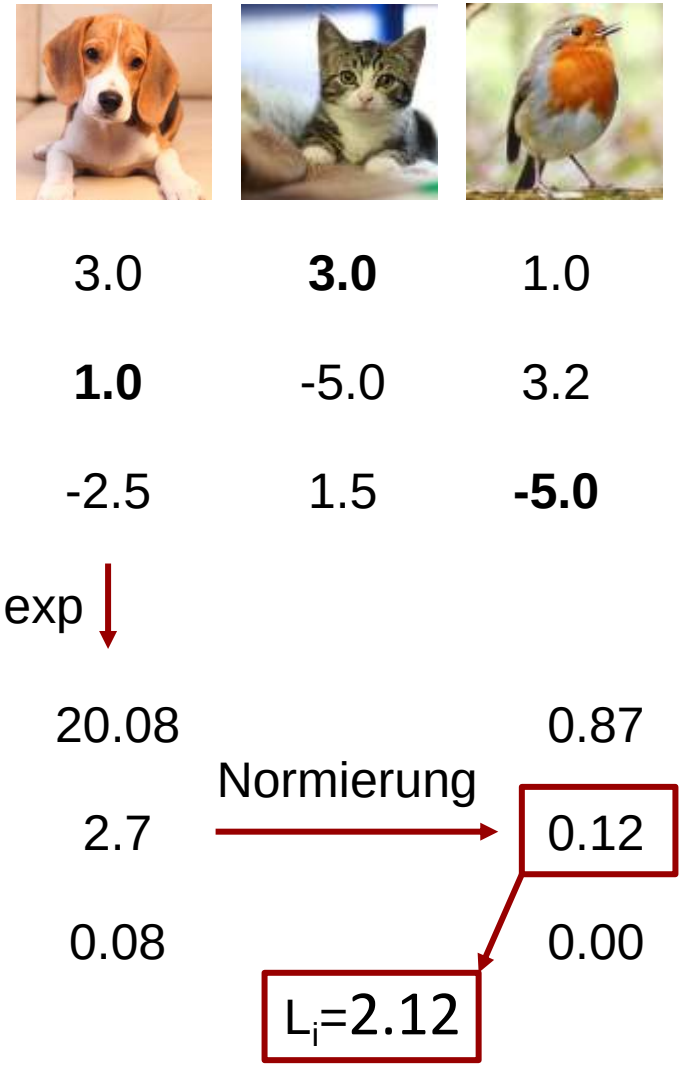
exp ↓

	20.08	0.87
Normierung →	2.7	0.12
	0.08	0.00

Kreuzentropie als Verlustfunktion

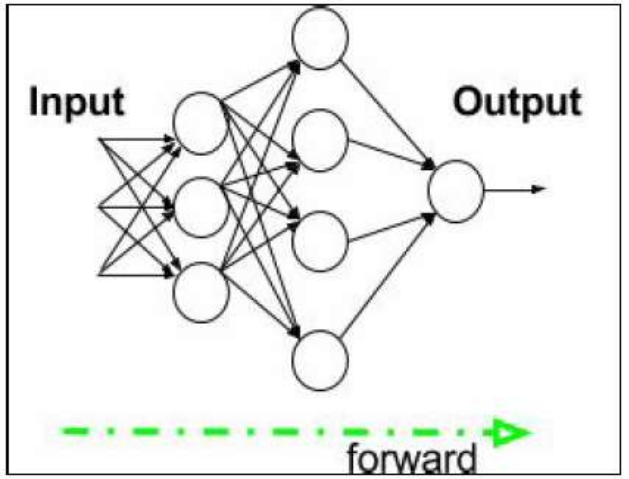
- Die **Kreuzentropie** eignet sich als Verlustfunktion für die Abweichung zweier Wahrscheinlichkeitsverteilungen
- Hat die Klasse y_i Wahrscheinlichkeit 1, entspricht die Kreuzentropie deren negativer Log-Likelihood. Zusammen mit der Softmax-Transformation ergibt sich

$$L_i = -\ln\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

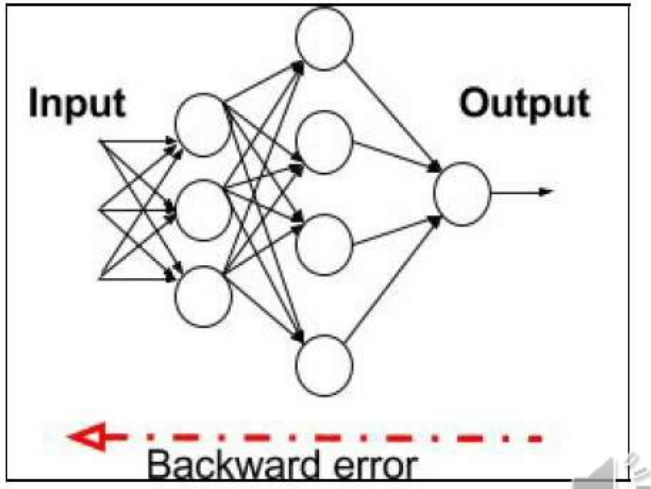


Grundidee der Backpropagation

Ziel: Gradientenabstieg erfordert Ableitung der Verlustfunktion nach den Netzwerkparametern



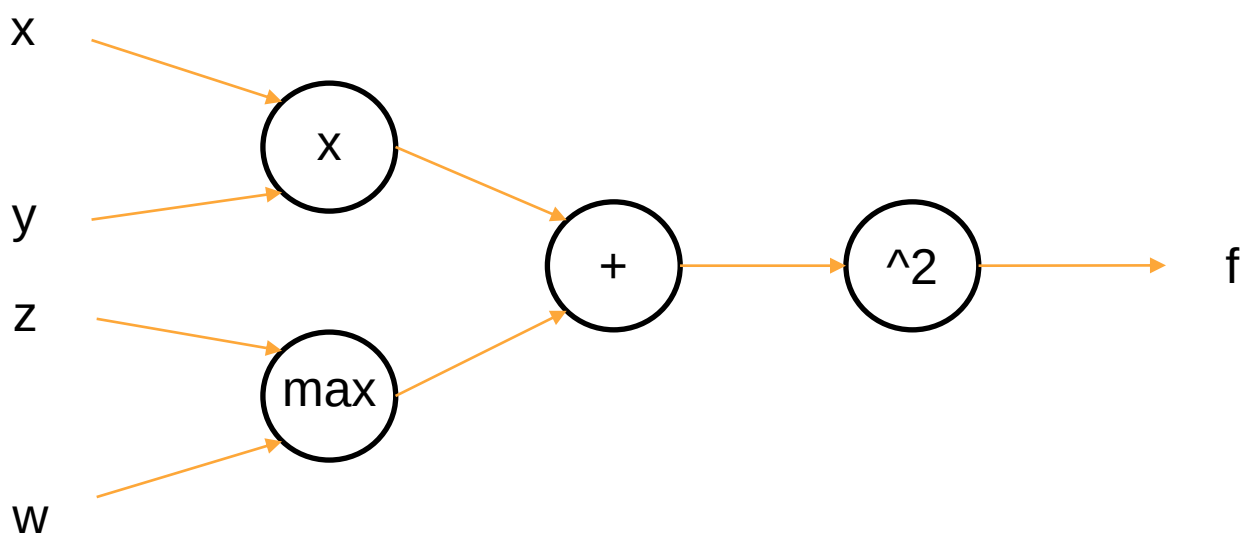
1. **Vorwärtspropagierung** berechnet die Ausgabe des Netzwerks und ermöglicht Berechnung der Verlustfunktion
   – Alle Zwischenergebnisse werden gespeichert



2. **Backpropagation** (Rückpropagierung) berechnet schrittweise, von hinten nach vorn, die gewünschten partiellen Ableitungen
   – Nutzt mittels Kettenregel die Zwischenergebnisse

Beispiel: Vorwärtspropagierung

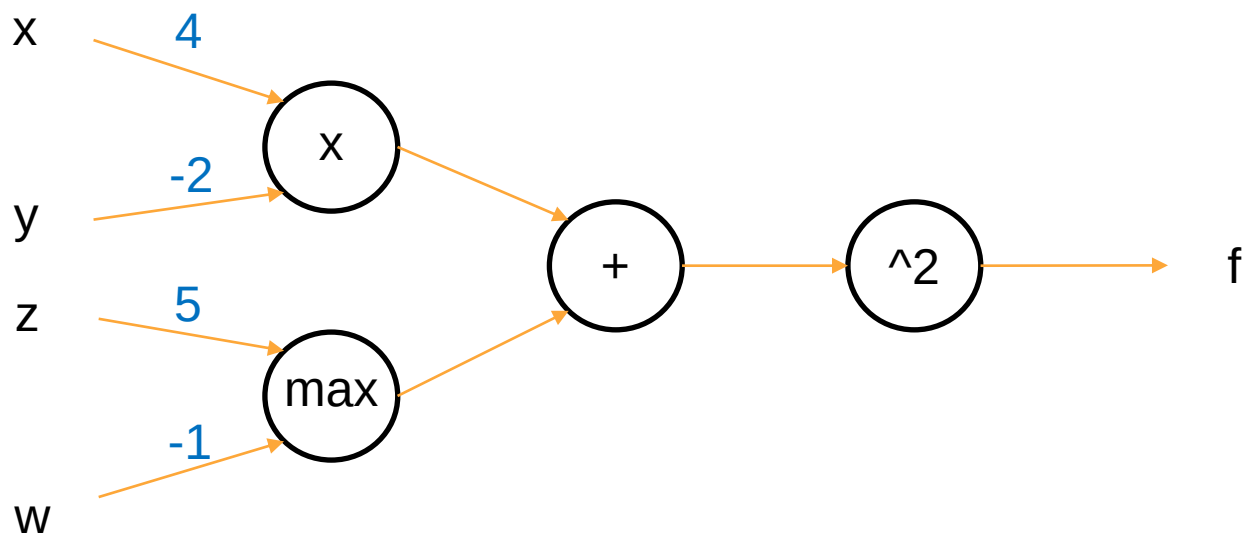
$f(x, y, z, w)$=(xy+max(z,w))^2



Beispiel: Vorwärtspropagierung

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

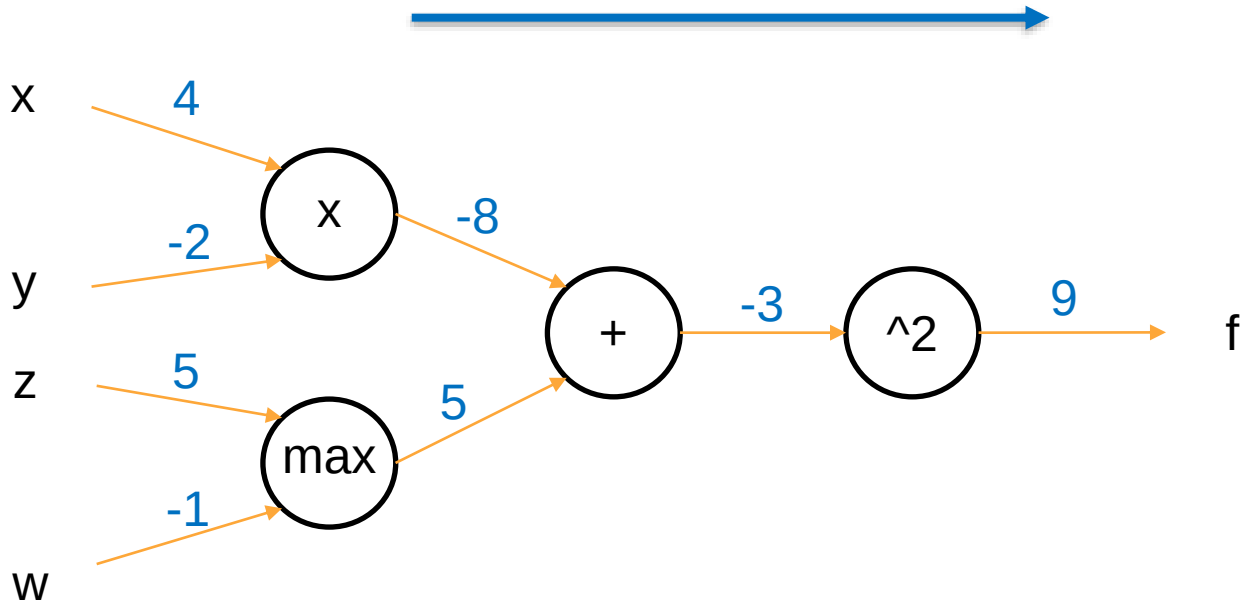
Vorwärtspropagierung von
x=4, y=-2, z=5, w=-1



Beispiel: Vorwärtspropagierung

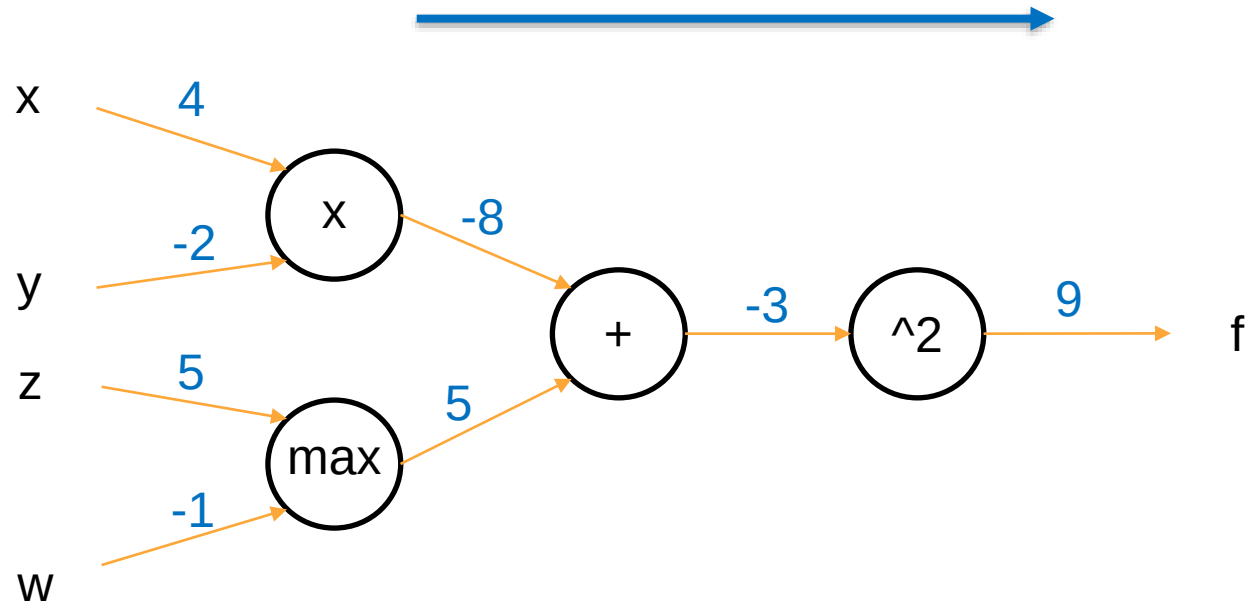
$f(x, y, z, w)$=(xy+max(z,w))^2

Vorwärtspropagierung von
x=4, y=-2, z=5, w=-1



Beispiel: Backpropagation

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



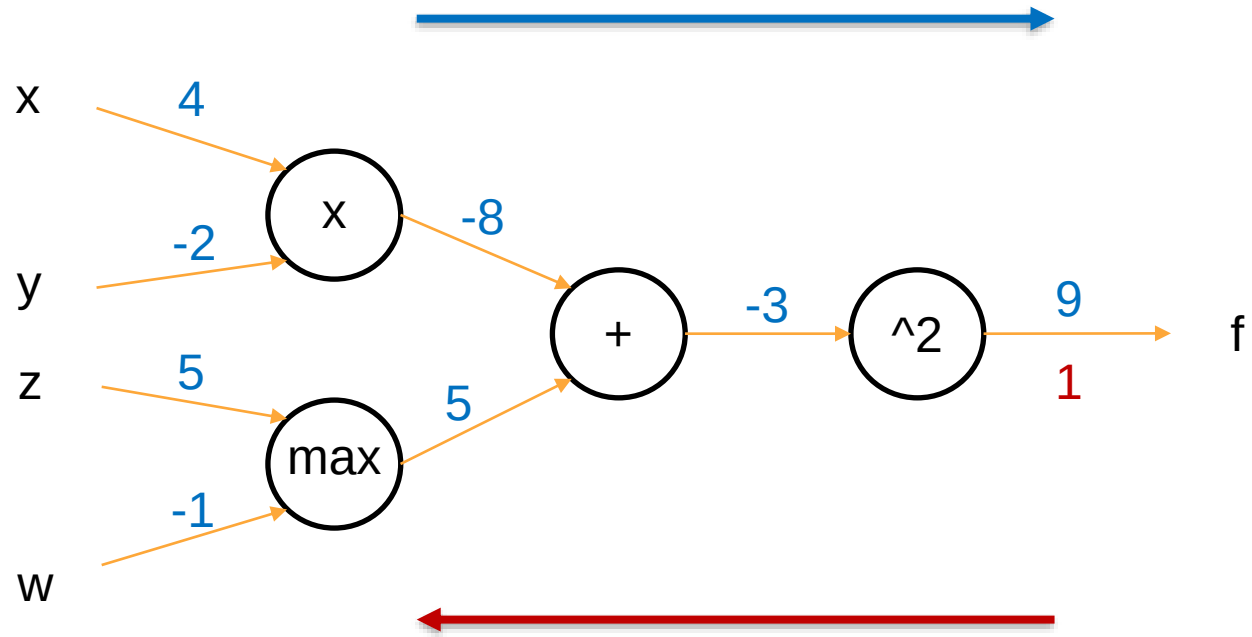
Vorwärtspropagierung von
x=4, y=-2, z=5, w=-1

Backpropagation:
Berechnet die Ableitung von f nach x, y, z, w

Hinweis: Beim Training neuronaler Netze nutzen wir das hier illustrierte Prinzip, um die Verlustfunktion nach den zu lernenden Parametern abzuleiten, insb. den Gewichten jeder Schicht

Beispiel: Backpropagation

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



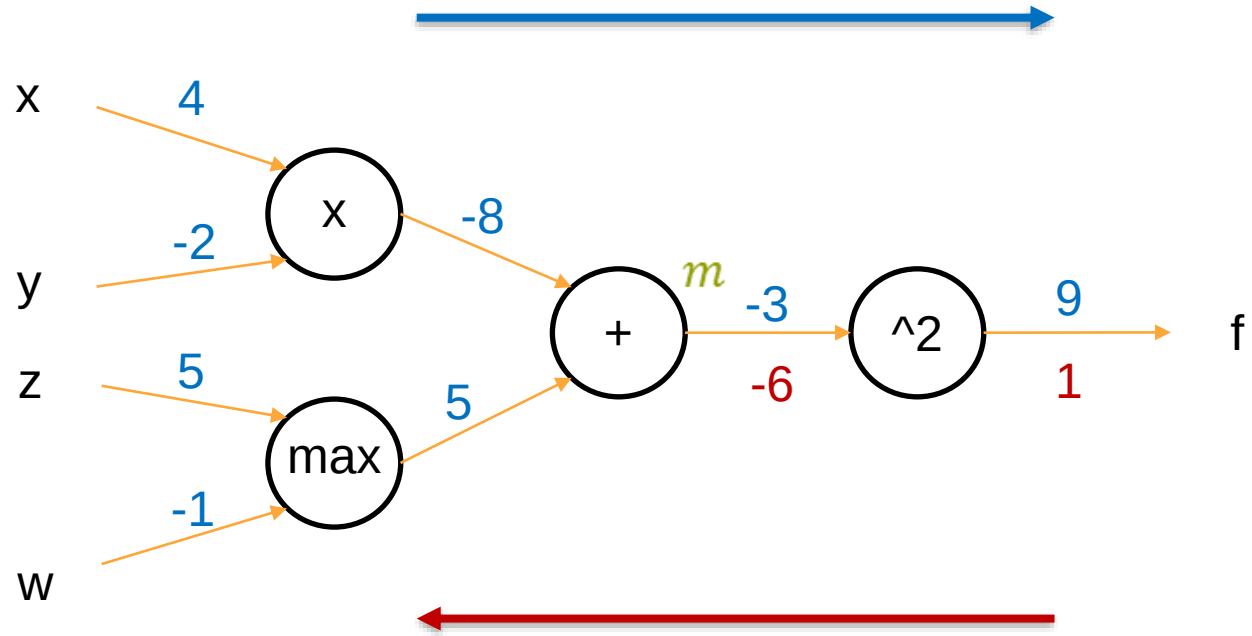
Vorwärtspropagierung von
x=4, y=-2, z=5, w=-1

Backpropagation:
Berechnet die Ableitung von f nach x, y, z, w

$$\frac{\partial f}{\partial f} = 1$$

Beispiel: Backpropagation

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



Vorwärtspropagierung von
x=4, y=-2, z=5, w=-1

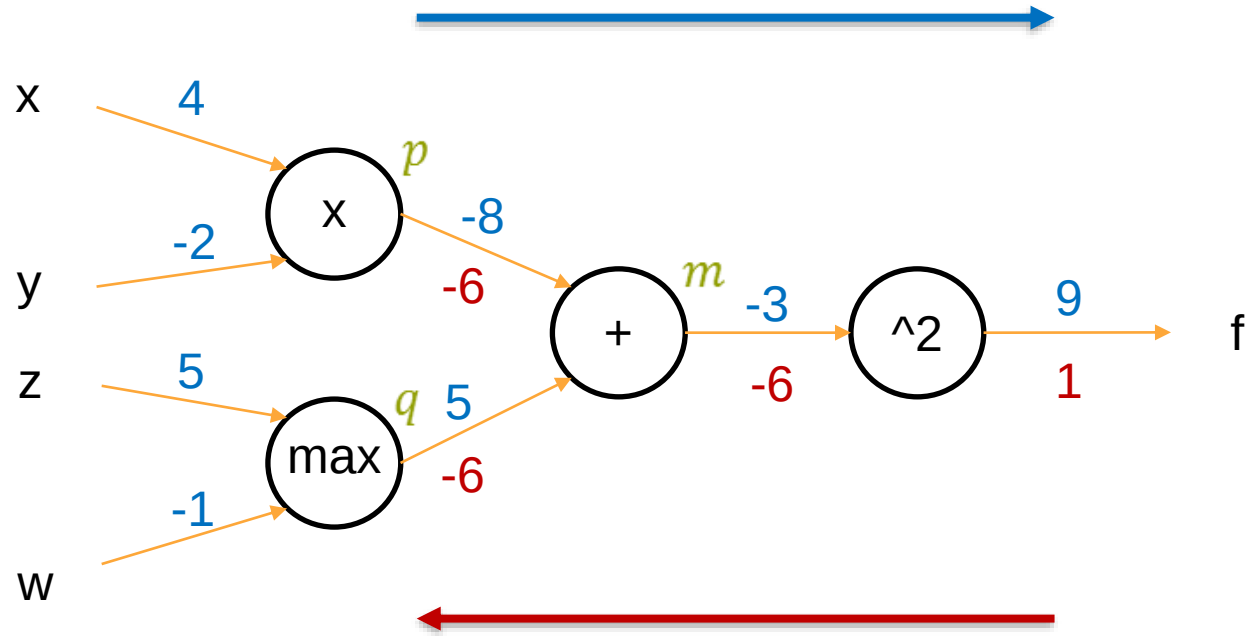
Backpropagation:
Berechnet die Ableitung von f
nach x, y, z, w

$$f = m^2$$

$$\frac{\partial f}{\partial m} = 2m$$

Beispiel: Backpropagation

$f(x, y, z, w)$=(xy+max(z,w))²



Vorwärtspropagierung von
x=4, y=-2, z=5, w=-1

Backpropagation:
Berechnet die Ableitung von f nach x, y, z, w

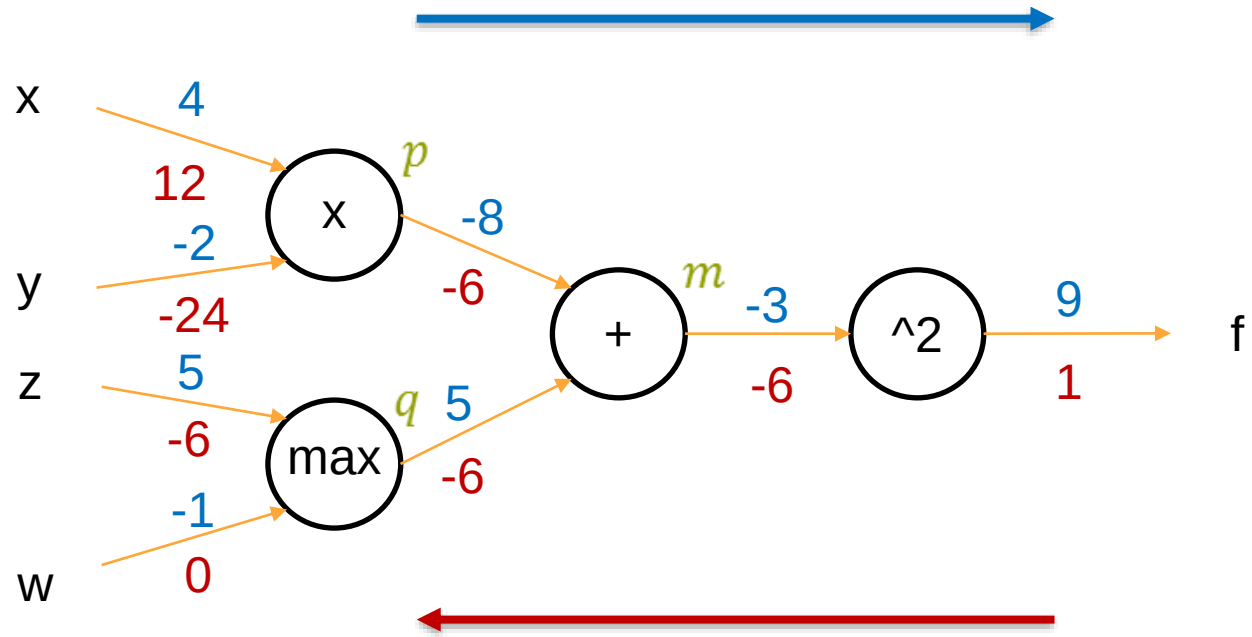
$$m = p + q$$

$$\frac{\partial f}{\partial p} = \frac{\partial f}{\partial m} \cdot \frac{\partial m}{\partial p}$$

Kettenregel

Beispiel: Backpropagation

$f(x, y, z, w)$=(xy+max(z,w))²



Vorwärtspropagierung von
x=4, y=-2, z=5, w=-1

Backpropagation:
Berechnet die Ableitung von f nach x, y, z, w

$$p = xy$$
$$q = \max(z, w)$$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial p} \cdot \frac{\partial p}{\partial x}, \qquad \frac{\partial f}{\partial y} = \frac{\partial f}{\partial p} \cdot \frac{\partial p}{\partial y}$$
$$\frac{\partial f}{\partial z} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial z}, \qquad \frac{\partial f}{\partial w} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial w}$$

Gradientenabstieg

- Ein **Gradientenabstieg** nutzt den per Backpropagation berechneten Gradienten $\nabla_{\mathbf{w}} L$ um die Gewichte $\mathbf{w}$ mit Lernrate λ in der Richtung anzupassen, die den Verlust L möglichst schnell verringert:

$$\mathbf{w} \mathrel{+}= -\lambda \nabla_{\mathbf{w}} L$$

- *Praktisches Problem*:
  - Die Berechnung der Verlustfunktion und ihrer Ableitungen ist sehr rechenaufwändig, da sie von allen Trainingsdaten abhängt
  - Gradientenabstieg macht relativ kleine Schritte und erfordert daher sehr häufige Auswertung

Stochastischer Gradientenabstieg

- **Stochastischer Gradientenabstieg** (SGD, *engl*. stochastic gradient descent) schätzt $\nabla_{\mathbf{w}} L$ in jedem Schritt mit einer zufälligen Teilmenge (“mini batch”) der Trainingsdaten
  - Richtung weniger zuverlässig, aber sehr viel schneller zu berechnen
- Eine **Epoche** bezeichnet eine bestimmte Zahl von Updates
  - *Traditionell*: Zahl der Trainingsdaten durch Minibatch-Größe, d.h. nach einer Epoche wurde (bei Ziehen ohne Zurücklegen) jedes Trainingsdatum einmal verarbeitet
  - *Manchmal auch:* Willkürlich festgelegte Zahl, z.B. 250 Updates

Gradientenabstieg mit Trägheit / Momentum

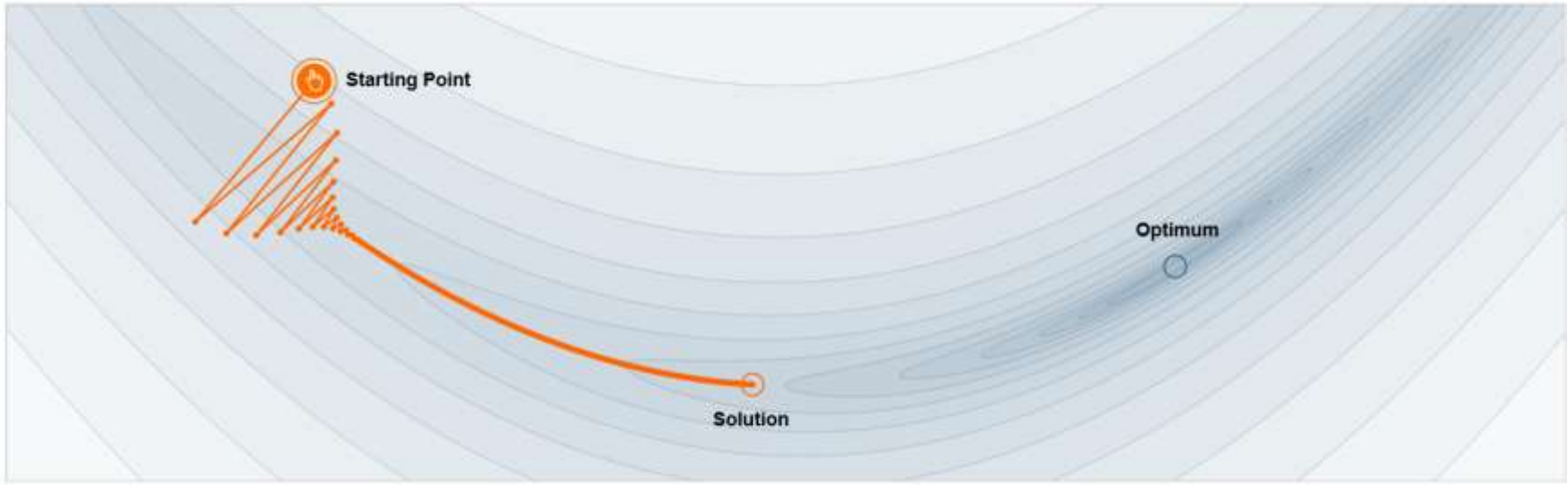
- *Problem*: Unzuverlässige Gradientenrichtungen wegen SGD und/oder starker Krümmung der Verlustfunktion
- *Idee*: Glättung durch Berechnung eines gleitenden Mittels der letzten Gradientenrichtungen
  - Schritte werden größer, wenn Gradienten in dieselbe Richtung zeigen
  - Ermöglicht außerdem Überwinden von Regionen schwacher Gradienten
  - *Analogie*: Rollen eines Balls über die „Landschaft“ der Verlustfunktion
    - Impuls (*engl*. momentum) ist proportional zur Geschwindigkeit
    - Impuls ändert sich durch Reibung und Schwerkraft
- **Momentum-Update** $\mathbf{w} \mathrel{+}= -\lambda\mathbf{v}$ mit Geschwindigkeit

$$\mathbf{v}^{(k)} := \mu\mathbf{v}^{(k-1)} + \nabla_{\mathbf{w}} L$$

  - Initialisierung mit $\mathbf{v}^{(0)} = \mathbf{0}$, Änderung bei jedem Update k
  - Abschwächung $0 \leq \mu < 1$ neben Lernrate λ weiterer Hyperparameter

Illustration: Gradienten-Abstieg vs Momentum

$$\mathbf{w} \mathrel{+}= -0.003 \nabla_{\mathbf{w}} L$$



$$\mathbf{w} \mathrel{+}= -0.003 \mathbf{v}$$
$$\mathbf{v}^{(k)} := 0.8 \mathbf{v}^{(k-1)} + \nabla_{\mathbf{w}} L$$

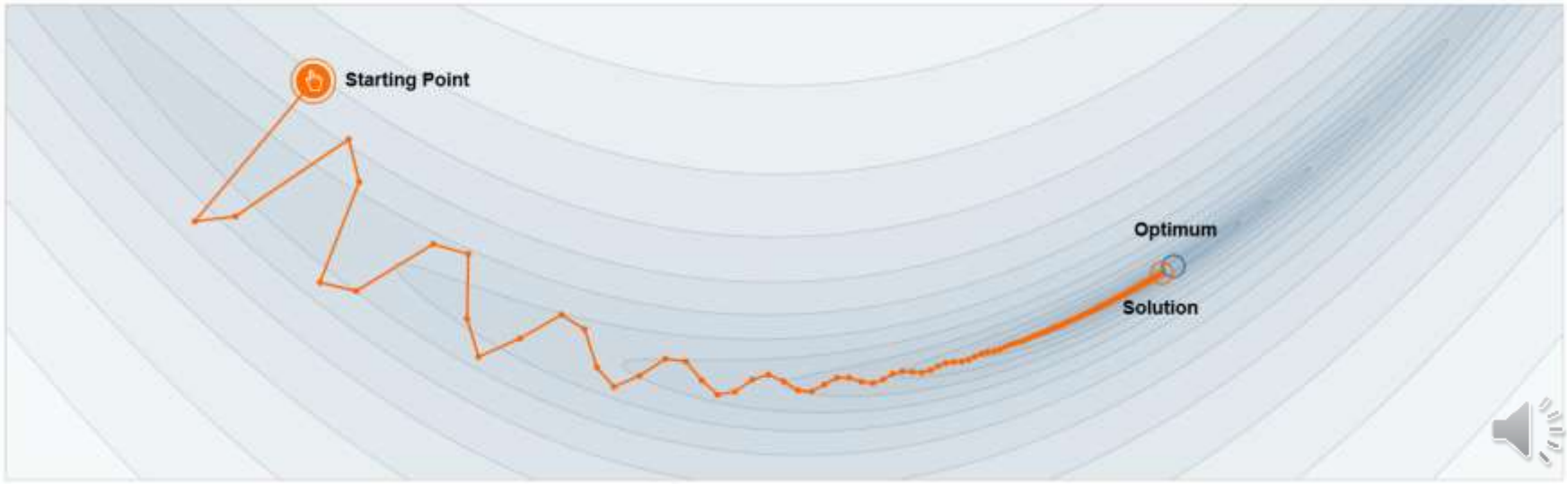
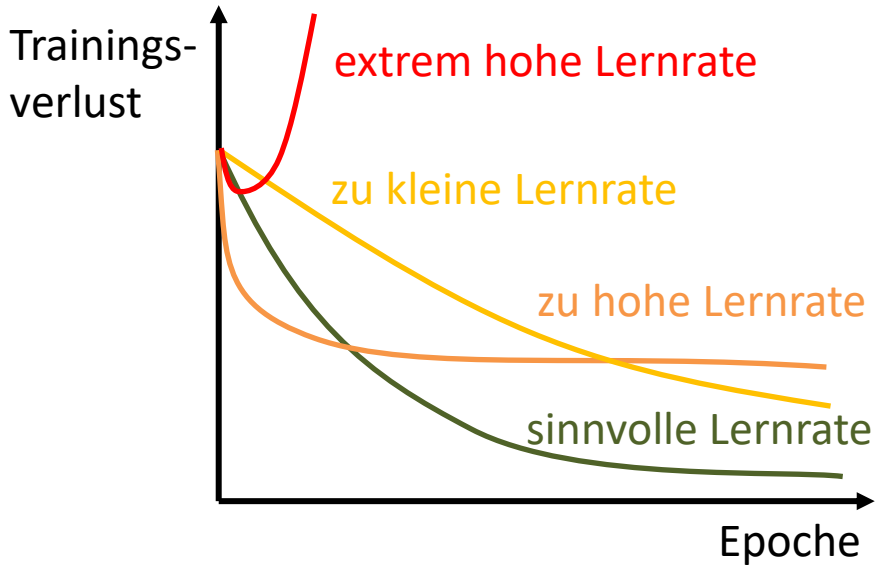


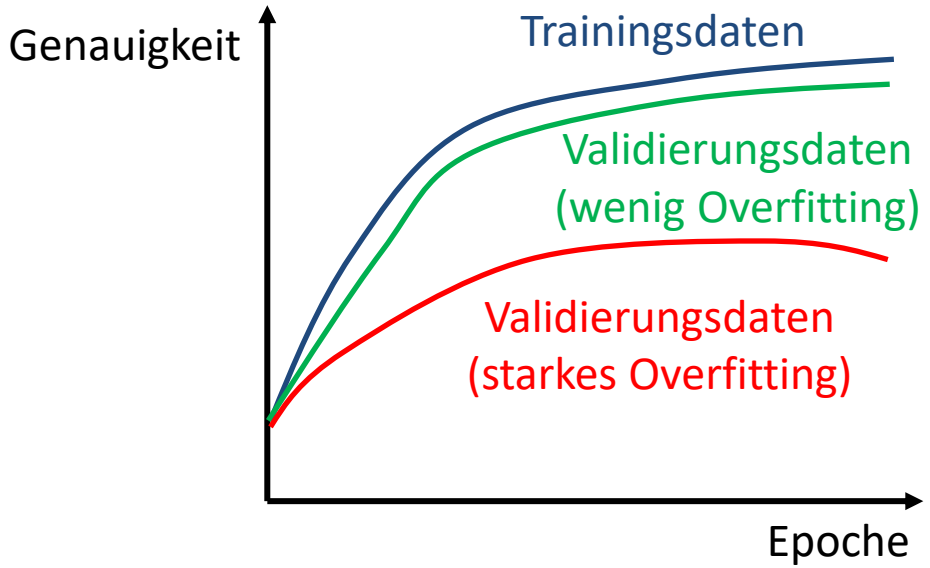
Illustration:
https://distill.pub/2017/momentum/

Überwachung des Lernprozesses

- Plotten des **Trainingsverlusts** zeigt, ob die Optimierung konvergiert oder Hyperparameter des Optimierers (z.B. Lernrate) angepasst werden müssen



- Ein Vergleich der Genauigkeit auf **Trainings-** und **Validierungsdaten** zeigt eine mögliche Überanpassung (overfitting)



Anpassung der Lernrate

- Die anfängliche Lernrate λ wird im Laufe des Trainings üblicherweise immer weiter verringert
- Verbreitete **Lernratenpläne** (*engl*. learning rate schedule):
  - **Schrittweise** Reduzierung um einen bestimmten Faktor, nach einer festen Zahl von Epochen oder wenn der Validierungsverlust stagniert
  - **Lineare** Reduzierung von einer anfänglichen Lernrate λ_0 auf eine ab der Iteration *k*=*T* gültige finale Lernrate λ_T

$$\lambda_k = (1-\alpha)\lambda_0 + \alpha\lambda_T \text{ mit } \alpha = \min\left(\frac{k}{T}, 1\right)$$

  - **Exponentielle** Reduzierung mit Zerfallsparameter γ

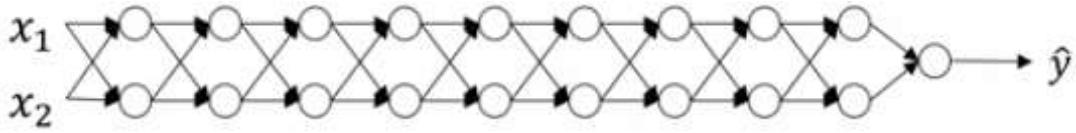
$$\lambda_k = \lambda_0 \mathrm{e}^{-\gamma k}$$

Adaptive Lernraten: AdaGrad und Adam

- *Beobachtung*: Das Update $\mathbf{w} \mathrel{+}= -\lambda \nabla_{\mathbf{w}} L$ verändert Parameter w_i mit einem höheren Einfluss auf die Verlustfunktion stärker
  - *Konsequenz*: Höchste Lernrate λ mit stabilem Training wird durch die „empfindlichsten“ Neuronen beschränkt. Andere Neuronen benötigen für angemessenen Fortschritt evtl. ein höheres λ
- **AdaGrad** berechnet Parameter-spezifische Lernraten
- **Adam** ist ein beliebtes Optimierungsverfahren für tiefe neuronale Netze. Es kombiniert adaptive Lernraten mit Momentum-Updates

Initialisierung der Gewichte flacher Netze

- **Mit Null:** Würde zu identischen Gradienten und Updates führen und ermöglicht daher kein sinnvolles Training!
  - *Aber*: Bias-Parameter b_k werden häufig auf Null initialisiert
- **Kleine Zufallszahlen,** z.B. $\mathcal{N}(0,0.01^2)$. Funktioniert für flache Netze, aber problematisch in tiefen Netzen
  - Vernachlässigen wir die Aktivierungsfunktion, multiplizieren sich die Gewichtsmatrizen aller Schichten:



$$\hat{y} = W_{l-1} W_{l-2} \dots W_2 W_1 X$$

  - Bei zu kleinen Gewichten „sterben“ die Aktivierungen aus
  - Erhöhung der Varianz birgt die Gefahr „explodierender“ Aktivierungen

Initialisierung der Gewichte tiefer Netze

- **Xavier et al.** initialisieren Gewichte eines Neurons mit $\mathcal{N}(0,1/m)$, wobei m die Zahl der Eingaben ist
  - *Idee*: Das Neuron berechnet $f\left(\sum_{j=1}^{m} w_{kj}x_j + b_k\right)$
  - Die Varianz sollte von Schicht zu Schicht erhalten bleiben. Bei der Addition unkorrelierter Zufallsvariablen addieren sich die Varianzen
- **He et al.** berücksichtigen außerdem den Effekt der Aktivierungsfunktion
  - Bei Verwendung von ReLU empfehlen sie daher Initialisierung mit $\mathcal{N}(0,2/m)$

Batch-Normalisierung

- Optimierung ändert alle Schichten des Netzwerks gleichzeitig
  - *Problem*: Durch das Update früherer Schichten ändern sich die Eingaben späterer Schichten. Deren Updates sind dadurch nicht mehr optimal.
  - Batch-Normalisierung **entkoppelt** die einzelnen Schichten
    - Aktivierungen **H** jeder Schicht werden zunächst standardisiert (Mittelwert 0, Varianz 1), dann mit zusätzlichen Parametern $\boldsymbol{\beta}$ und $\boldsymbol{\gamma}$ verschoben und skaliert
    - Mittelwert und Varianz der Aktivierungen hängen direkt von $\boldsymbol{\beta}$ und $\boldsymbol{\gamma}$ ab, nicht mehr vom komplexen Zusammenspiel aller vorherigen Gewichte
- *Vorteile:* Ermöglicht höhere Lernraten, reduzierte Abhängigkeit von der Initialisierung, häufig bessere Ergebnisse
- *Hinweis*: Zusätzlich zur Batch-Normalisierung ist es üblich die Eingaben des Netzwerks zu standardisieren

Regularisierung

Um tiefe neuronale Netze zu **regularisieren** und Überanpassung (overfitting) zu reduzieren nutzt man u.a.

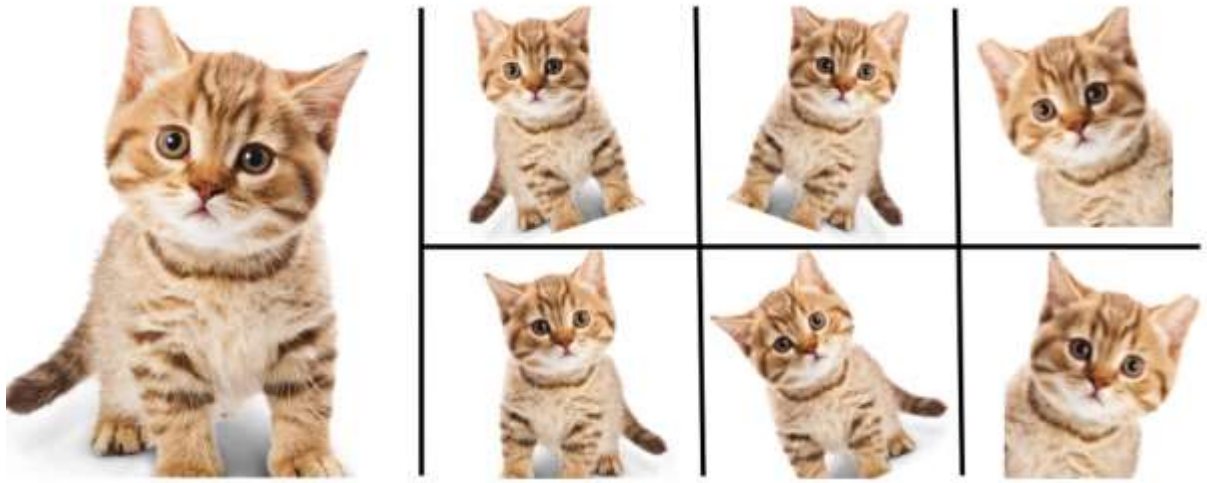
- **Bestrafung großer Parameterwerte** durch modifizierte Zielfunktion:

$$L = \underbrace{\frac{1}{N}\sum_i L_i}_{\text{Datenterm}} + \underbrace{\nu R(W)}_{\text{Regularisierer}} \quad \text{z.B. mit } R(W) = \sum_l \sum_k \sum_j w_{lkj}^2$$

  - w_{lkj} ist das *j*-te Gewicht des *k*-ten Neurons der *l*-ten Schicht
- **Early Stopping:** Nutzt die Parameter der Epoche, nach der der Validierungsfehler am geringsten war, selbst wenn der Trainingsfehler danach weiter gesunken ist
- **Dropout**: Zufälliges Auslassen von Neuronen während des Trainings mit einstellbarer Wahrscheinlichkeit *p*
  - Wird bei der Anwendung auf Testdaten kompensiert, indem die Ausgaben verborgener Schichten mit *p* skaliert werden

Datenaugmentierung

- Ein **größerer Trainingsdatensatz** ist häufig das erfolgreichste Mittel um eine bessere Generalisierung zu erreichen
  - ...aber die Aufnahme/Annotation ist leider meist sehr aufwändig
- **Datenaugmentierung** erzeugt künstlich zusätzliche Daten
  - modifiziert vorhandene Bilder so, dass die Labels gültig bleiben oder automatisch angepasst werden können
  - Beispiele:
    - Verschiebung, Rotation
    - Ausschnitte, Spiegelung
    - (Leichte) Deformationen
    - Farb- und Kontraständerungen



Bildquelle: nanonets.com

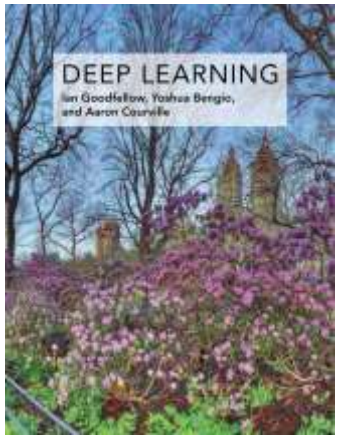
Zusammenfassung

Das **Training neuronaler Netze** erfordert

- die Wahl einer geeigneten **Verlustfunktion**
- die Berechnung ihres Gradienten bezüglich der Netzwerk-Parameter mittels **Backpropagation**
- die Wahl eines geeigneten **Optimierers** und seiner Hyperparameter, insbesondere **Lernrate** und **Lernratenplan**
  - *beliebte Tricks:* **Momentum** und **adaptive Lernraten**
  - **Batch-Normalisierung** vereinfacht die Suche nach geeigneten Parametern
- die Überwachung des Lernprozesses und den Einsatz geeigneter **Regularisierung** und **Augmentierung**

Zum Nach- und Weiterlesen

- Ian Goodfellow, Yoshua Bengio, Aaron Courville: “Deep Learning.” MIT Press, 2016
  https://www.deeplearningbook.org/



- Christopher Bishop with Hugh Bishop: “Deep Learning. Foundations and Concepts.” Springer, 2024
  https://www.bishopbook.com/