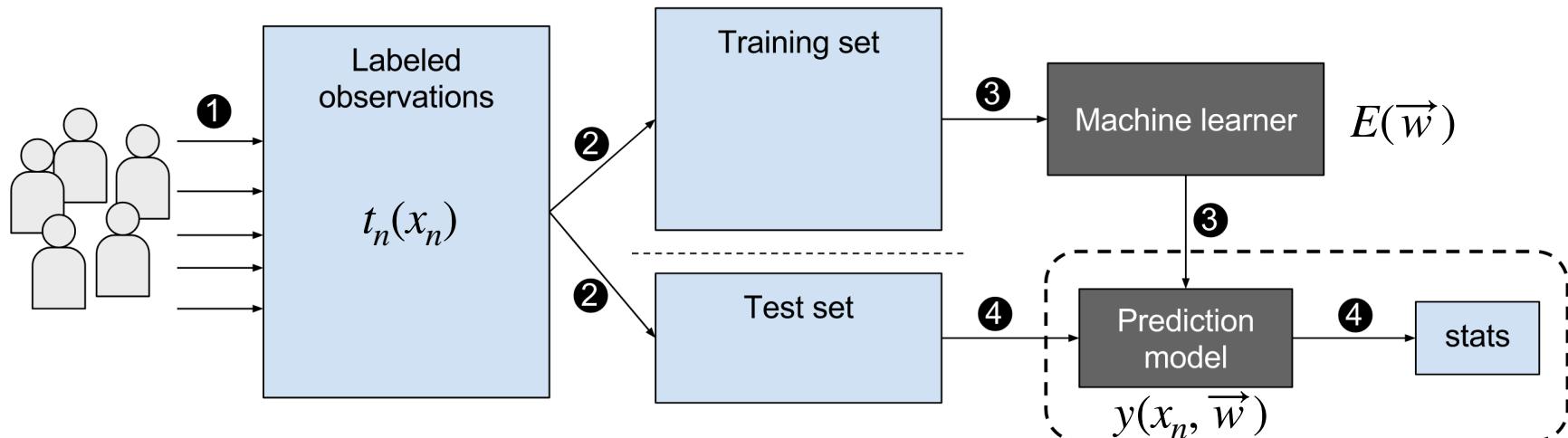


Deep and convolutional neural networks

Konstanz, 10.06.2024

Überwachtes Lernen

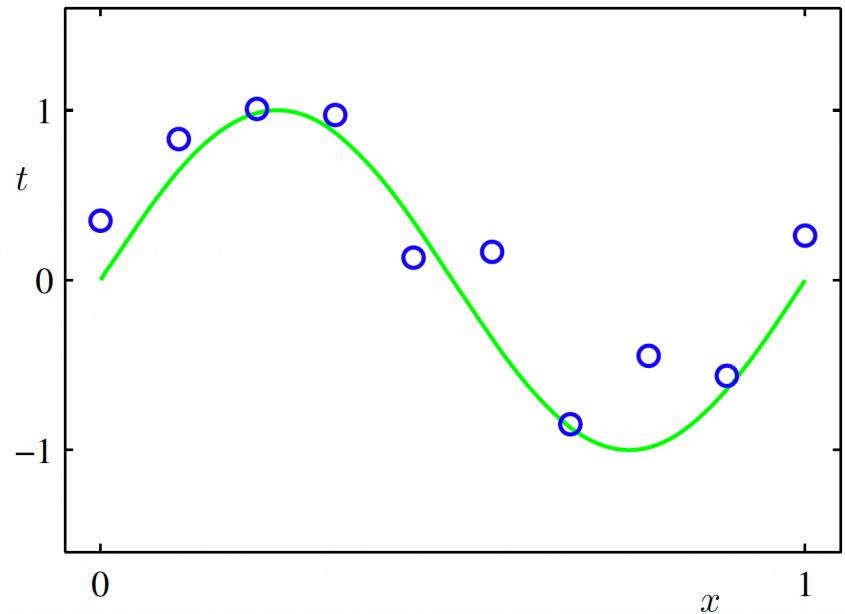
- Eingangsobjekte und ein gewünschter Ausgabewert trainieren das Modell
- Problemlösung mit Überwachtem Lernen:
 - Trainingsset sammeln, das repräsentativ für die zu lösenden Aufgabe ist
 - Transformation der Eingangsobjekte in einen Feature-Vektor, der die Merkmale des Objekts beschreibt
 - Modell wählen, welches das Problem lösen soll
 - Bestimmung der Parameter des Modells mit einem Lernalgorithmus
 - Genauigkeit des Modells bewerten, indem es auf einem separaten Testset gemessen wird



Wikimedia Commons https://commons.wikimedia.org/wiki/File:Supervised_machine_learning_in_a_nutshell.svg 2016

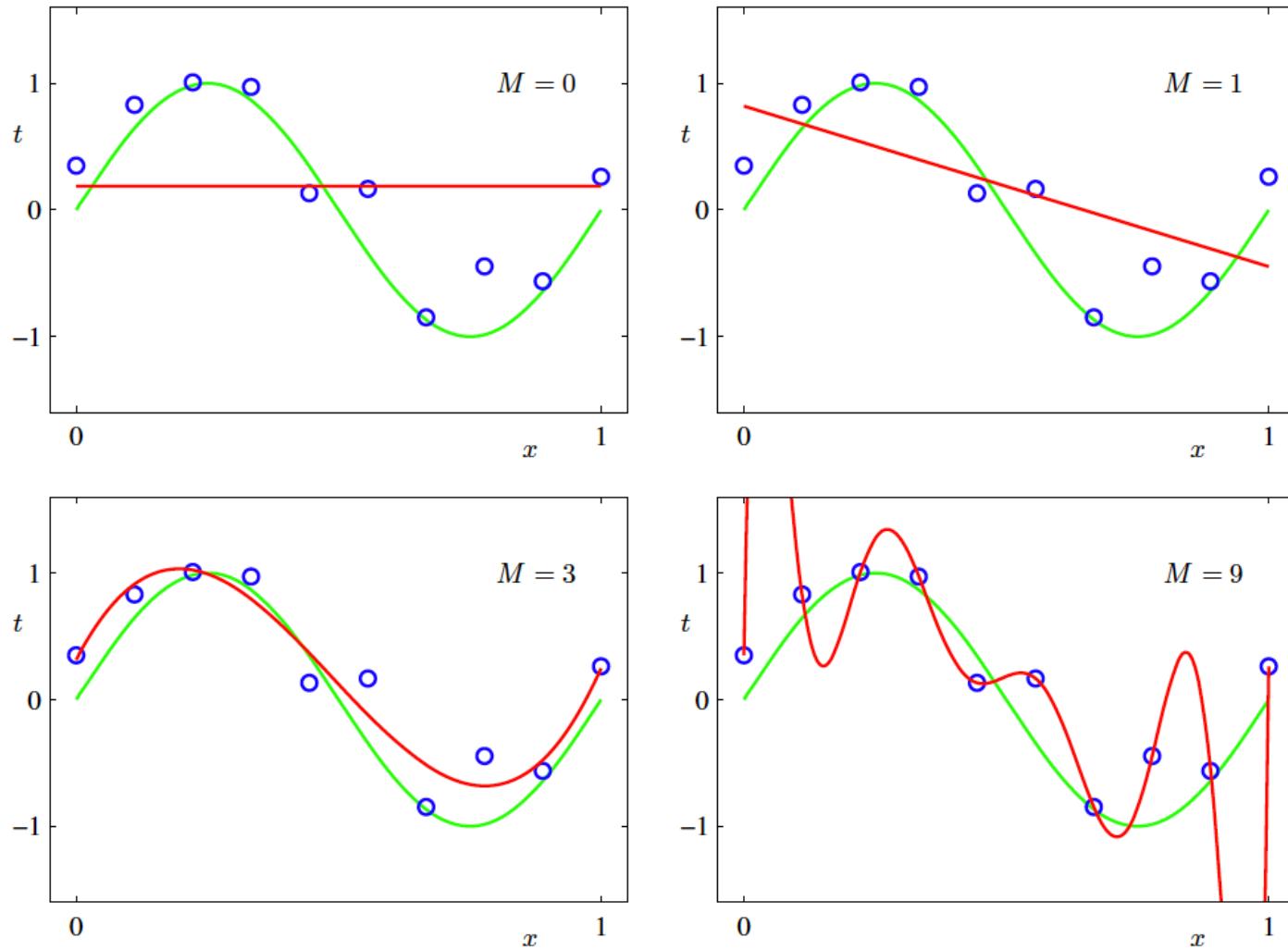
Beispiel:

- exakte, unbekannte Funktion $h(x) = \sin(2\pi x)$; Trainingsdaten $t_n(x_n)$ mit $n \in N$
- Polynomfunktion als Modell Fkt.: $y(x, \vec{w}) = \sum_{j=0}^M w_j x^j$
- Minimierung von $E(\vec{w}) = \frac{1}{2} \sum_{n=1}^N \{y(x_n, \vec{w}) - t_n\}^2$ (Residuenquadratsumme)
 $\Rightarrow E_{RMS} = \sqrt{(2E(w)/N)} = \sqrt{E_{MSE}}$
- $y(x_n, \vec{w})$ wird durch Minimierung von $E(\vec{w})$ bestimmt
- Frei ist noch Wahl von M (*model selection*)



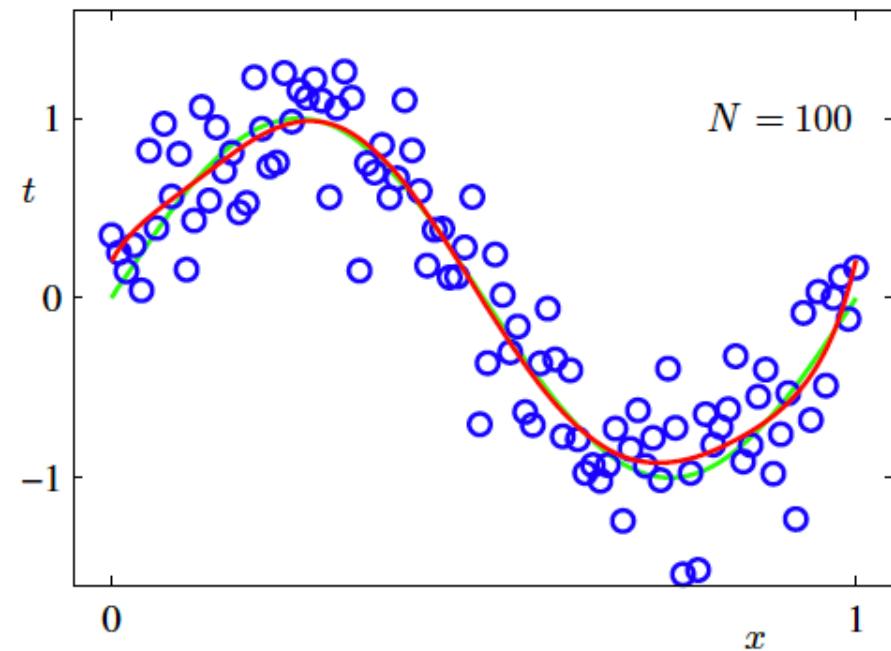
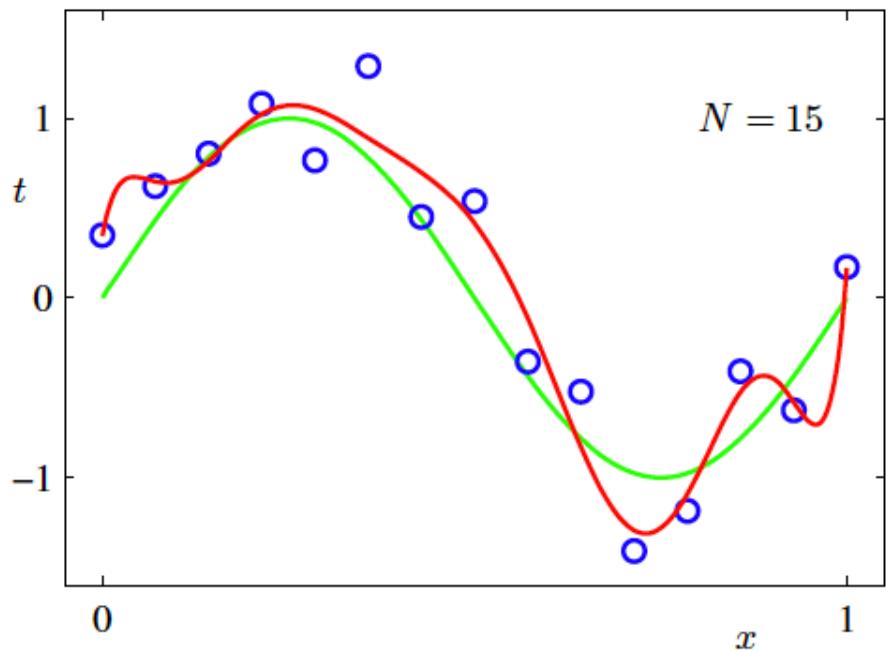
M. Jordan et al. / Pattern Recognition And
Machine Learning (2006)

Beispiel: Overfitting



M. Jordan et al. / Pattern Recognition And Machine Learning (2006)

Beispiel: Abhangigkeit Trainingsdaten



M. Jordan et al. / Pattern Recognition And Machine Learning (2006)

- Gefittete Funktion $y(x, \vec{w})$ ist abhangig von (Groe) der Trainingsdaten
- Fur $M < N$ scheint overfitting vermieden zu werden
- Vergleich mit Polynomfunktion davor: Beide Funktionen variieren stark bei verschiedenen Datensatzen!

Bias-variance trade-off

- Bias-Fehler:
 - Entsteht durch zu einfache Annahmen über die Beziehung zwischen den Eingabemerkmale und den Zielvariablen macht.
 - Hoher Bias Fehler: Unteranpassung
- Varianz:
 - Empfindlichkeit gegenüber Schwankungen bei verschiedenen Trainingsdatensätzen
 - Hohe Varianz: Modell modelliert zufällige Rauschen in den Trainingsdaten, kleine Änderungen in den Trainingsdaten führen zu stark unterschiedlichen Vorhersagen, Überanpassung
- Zu hohe Modellkomplexität:
 - kleiner Bias-Fehler, große Varianz
 - Trainingsdaten werden gut dargestellt, Vorhersage auf anderen Daten oft weniger genau
- Zu geringe Modellkomplexität:
 - großer Bias-Fehler, geringe Varianz
 - Trainingsdaten nicht gut modelliert, wichtige Gesetzmäßigkeiten nicht erfasst

Bias-variance trade-off

- Der Mean Squared Error (MSE) lässt sich in einen Bias-Fehler und einen Varianz-Fehler zerlegen

$$\bullet \quad \{y(x, D) - h(x)\}^2$$

$$= \left\{ y(x, D) - \mathbb{E}_D [y(x, D)] + \mathbb{E}_D [y(x, D)] - h(x) \right\}^2$$

$$= \{y(x, D) - \mathbb{E}_D [y(x, D)]\}^2 - \{\mathbb{E}_D [y(x, D)] - h(x)\}^2$$

$$+ 2 \{y(x, D) - \mathbb{E}_D [y(x, D)]\} \{\mathbb{E}_D [y(x, D)] - h(x)\}$$

$$\overbrace{\mathbb{E}_D [y(x, D) - \mathbb{E}_D [y(x, D)]]} = 0$$

$$\Rightarrow \mathbb{E}_D [\{y(x, D) - h(x)\}^2] = \mathbb{E}_D [\{\mathbb{E}_D [y(x, D)] - h(x)\}^2] + \mathbb{E}_D [\{y(x, D) - \mathbb{E}_D [y(x, D)]\}^2]$$

$$= \underbrace{\{\mathbb{E}_D [y(x, D)] - h(x)\}^2}_{\text{Bias}^2} + \underbrace{\mathbb{E}_D [\{y(x, D) - \mathbb{E}_D [y(x, D)]\}^2]}_{\text{Variance}}$$

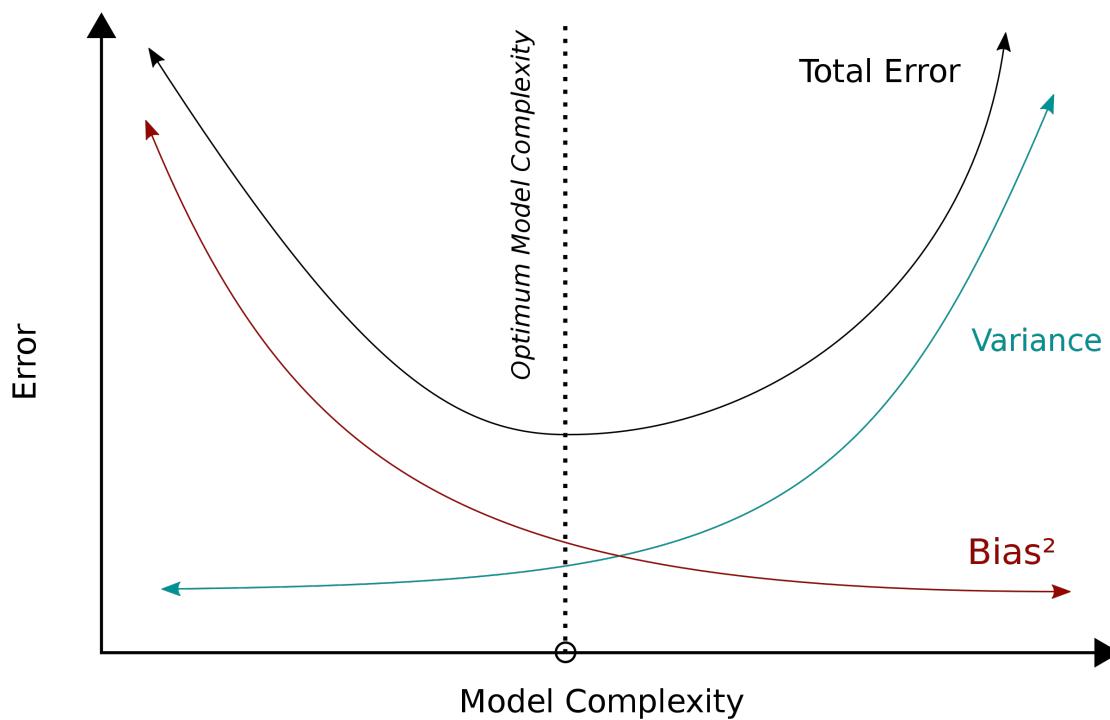
Notation:

$h(x)$: exakte, nicht bekannte, Regressionsfunktion

\mathbb{E}_D : Erwartungswert bezüglich aller Trainingsdatensätze $D = \{(x_1, t_1), \dots, (x_n, t_n)\}$

$y(x, D)$: Vorhersagefunktion des Lernalgorithmus zum Trainingsdatensatz D

Bias-variance trade-off



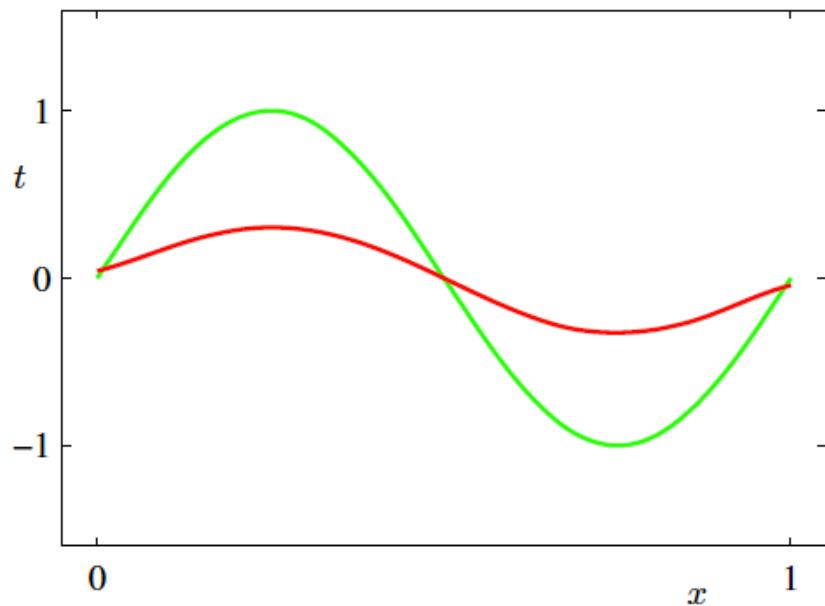
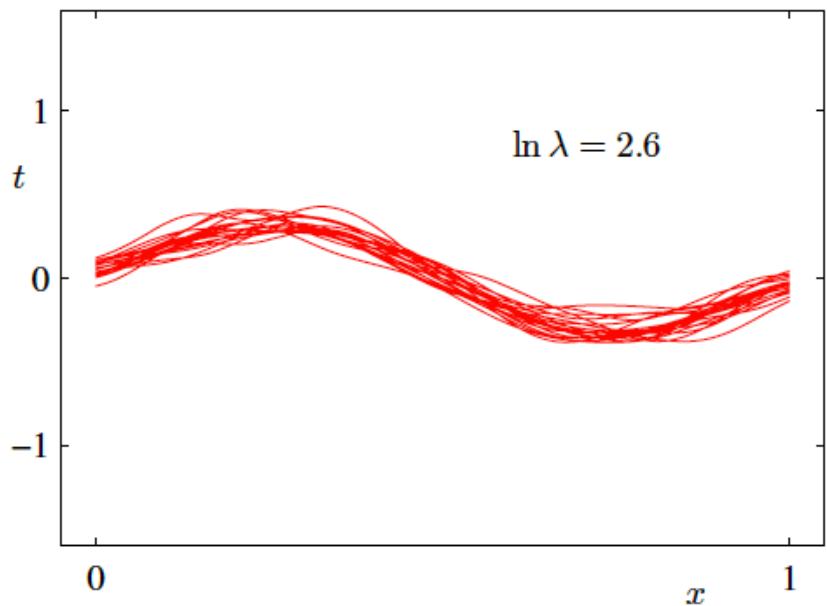
Wikimedia commons [https://commons.wikimedia.org/
wiki/File:Bias_and_variance_contributing_to_total_error.svg](https://commons.wikimedia.org/wiki/File:Bias_and_variance_contributing_to_total_error.svg) 2021

- **Bias-Varianz-Dilemma:** Bias und Varianz können nicht gleichzeitig minimiert werden

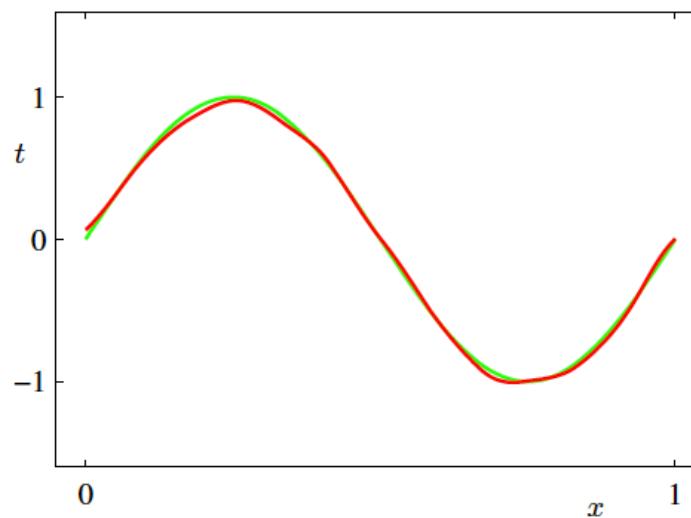
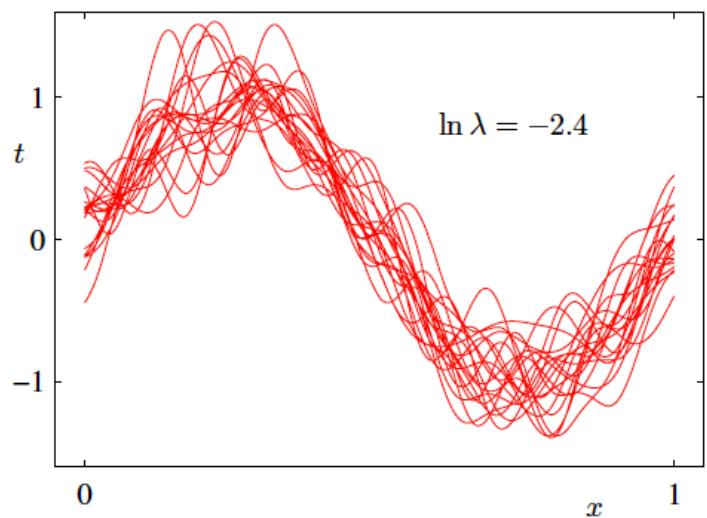
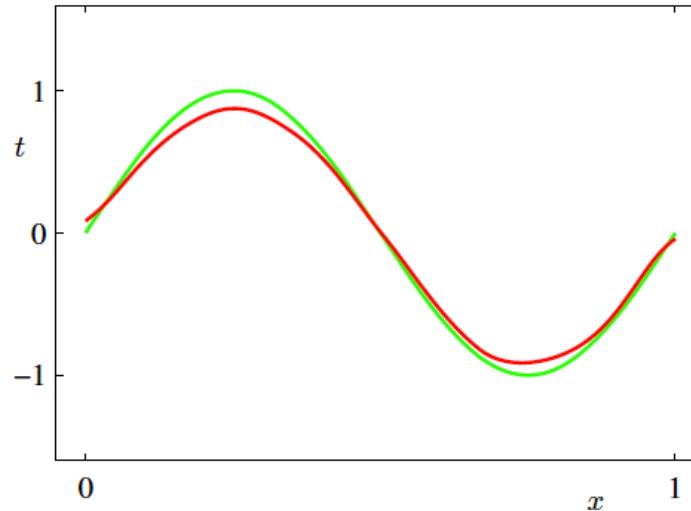
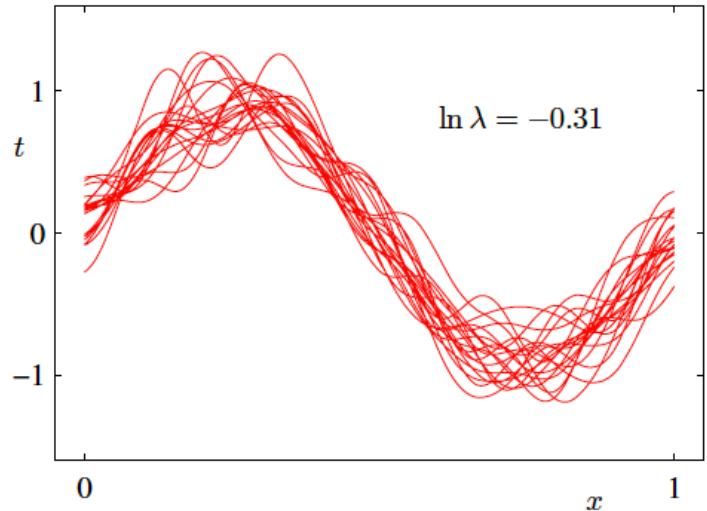
Beispiel: Bias-variance trade-off

Polynomfunktion als Modell Fkt.: $y(x, \vec{w}) = \sum_{j=0}^M w_j x^j$

$$E(\vec{w}) = \frac{1}{2} \sum_{n=1}^N \{y(x_n, \vec{w}) - t_n\}^2 + \frac{\lambda}{2} \|\vec{w}\|^2 \quad \text{für } \lambda = 0 \quad \Rightarrow \text{Residuenquadratsumme}$$



Beispiel: Bias-variance trade-off

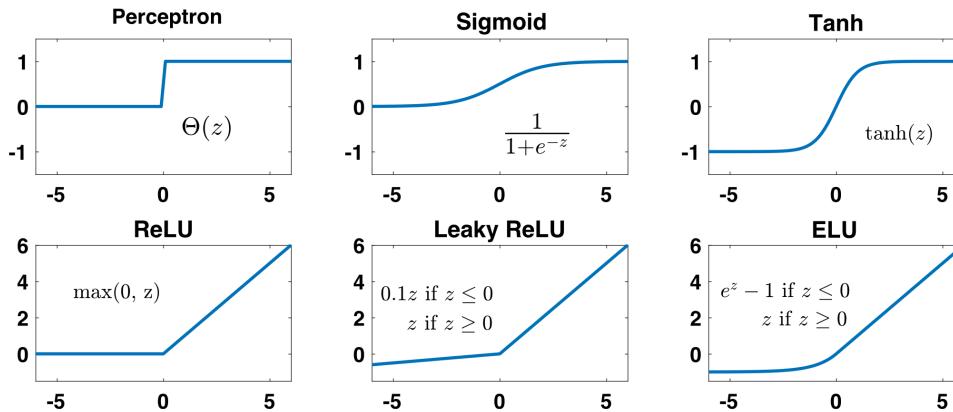


Feedforward Netzwerke (fully connected layers)

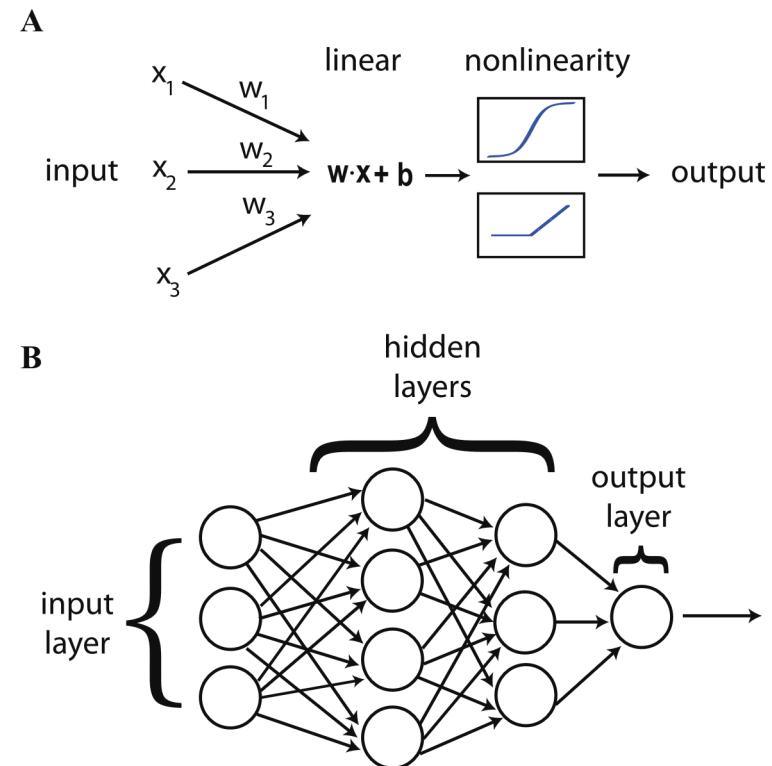
- Input $\mathbf{x}^{(l-1)}$, Gewichte $\mathbf{W}^{(l)}$ und Biases $\mathbf{b}^{(l)}$

- Output $a_i^l = f \left(\left(\sum_j W_{ij}^{(l)} \cdot x_j^{(l-1)} \right) + b_i^{(l)} \right)$

- Nicht lineare Aktivierungsfunktionen $f(\cdot)$ für die Neuronen



P. Mehta et al. / Phys. Rep. (2019) 810, 1-124



P. Mehta et al. / Phys. Rep. (2019) 810, 1-124

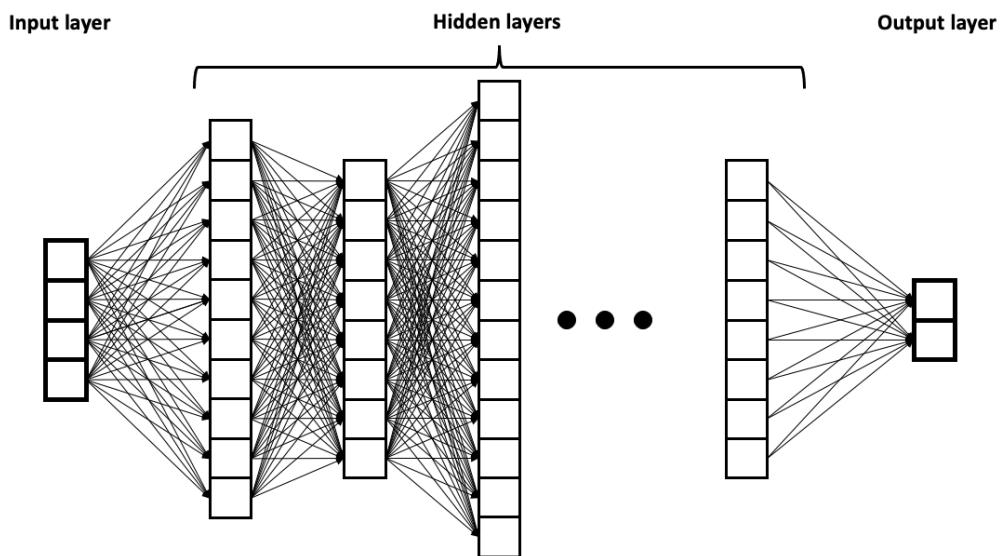
- Nicht saturierende Funktionen haben sich als erfolgreicher erwiesen

Feedforward Netzwerke

- Oft mit Softmaxfunktion

$$f(\mathbf{a})_j = \frac{e^{a_j}}{\sum_{i=1}^M e^{a_i}}, j = 1, \dots, M$$

als Aktivierungsfunktion der Output-Layer bei Klassifizierungsproblemen



<https://commons.wikimedia.org/wiki/>.

File:Example_of_a_deep_neural_network.png

Algorithmus 1 Feedforward Algorithmus

Require: L : Anzahl der Schichten.

Require: $\mathbf{b}^{(l)}$, $l \in \{1, \dots, L\}$: Biases des Modells.

Require: $\mathbf{W}^{(l)}$, $l \in \{1, \dots, L\}$: Gewichtungsmatrizen des Modells.

Require: \mathbf{x} zu verarbeitender Input des Modells.

```
1:  $\mathbf{a}^{(0)} = \mathbf{x}$ 
2: for  $l = 1, \dots, L$  do
    $\mathbf{z}^{(l)} = \mathbf{W}^{(l)} \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}$ 
    $\mathbf{a}^{(l)} = f(\mathbf{z}^{(l)})$ 
3: end for
```

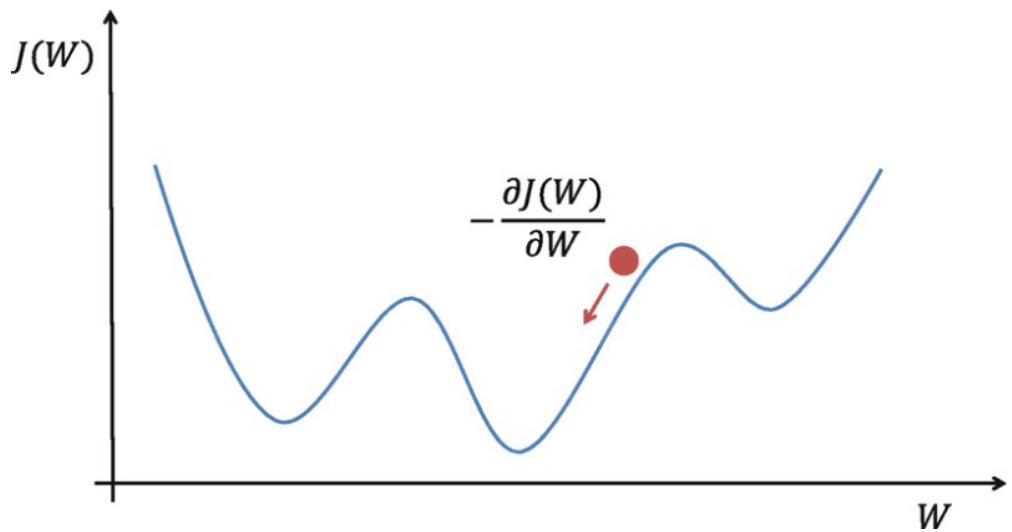
Training bei Deep Learning

- Kreuzentropie Kostenfunktion mit Modellparametern θ :

$$C(\theta, \mathbf{x}) = - \sum_i y_i(\mathbf{x}) \log (\hat{y}_i(\theta, \mathbf{x})) + (1 - y_i(\mathbf{x})) \log (1 - \hat{y}_i(\theta, \mathbf{x})), \text{ mit } y_i \in \{0,1\}$$

und output $\hat{y}_i = p(y_i = 1 | \theta, \mathbf{x})$ für Datenpunkt \mathbf{x}

- Gradient Descent: Passe Parameter der Gewichtungsmatrizen $\mathbf{W}^{(l)}$ und Biases $\mathbf{b}^{(l)}$ mit Lernrate η entgegen der Richtung des Gradienten der Kostenfunktion an



$$W_{jk}^{(l), \text{neu}} = W_{jk}^{(l), \text{alt}} - \eta \frac{\partial C(\theta)}{\partial W_{jk}^{(l)}} \Bigg|_{\theta^{\text{alt}}}$$

$$b_j^{(l), \text{neu}} = b_j^{(l), \text{alt}} - \eta \frac{\partial C(\theta)}{\partial b_j^{(l)}} \Bigg|_{\theta^{\text{alt}}}$$

- Mehrere Wiederholungen (epochs)
- Gradienten Berechnen:

Back-propagation Algorithmus

Varianten des Gradient-Descent

- **Mini-Batch Gradient Descent:** Zufällige Einteilung der Trainingsdaten in Minibatches bei jedem Trainingsdurchlauf $\theta = \theta - \frac{\eta}{n} \sum_{i=1}^n \nabla C_i(\theta)$, wobei $C_i(\theta)$ der Wert der Kostenfunktion zur i -ten Minibatch aus dem Trainingsdatenset ist.
- Geringere Wahrscheinlichkeit für die Konvergenz zu einem lokalen Minimum.
- **Adam Optimizer:** Basiert auf Berechnung der ersten beiden Momente der Gradienten, zur t -ten Trainingsiteration.
- Aktualisierungsregel für einen Parameter θ mit exponentiellen Abklingfaktoren β_1 (typischer Wert $\beta_1 = 0.9$) des ersten und β_2 (typischer Wert $\beta_2 = 0.999$) des zweiten Moments:

$$m_\theta^{(t+1)} = \beta_1 m_\theta^{(t)} + (1 - \beta_1) \nabla_\theta C^{(t)} \quad (\text{Erstes Moment})$$

$$v_\theta^{(t+1)} = \beta_2 v_\theta^{(t)} + (1 - \beta_2) (\nabla_\theta C^{(t)})^2 \quad (\text{Zweites Moment})$$

$$\hat{m}_\theta = \frac{m_\theta^{(t+1)}}{1 - \beta_1} \quad \hat{v}_\theta = \frac{v_\theta^{(t+1)}}{1 - \beta_1} \quad \theta^{(t+1)} = \theta^{(t)} - \eta \frac{\hat{m}_\theta}{\sqrt{\hat{v}_\theta} + \epsilon}$$

- Konvergiert schneller als Mini-Batch Gradient-Descent

Back-propagation Algorithmus

- Output Layer L: $\Delta_j^{(L)} = \frac{\partial C}{\partial z_j^{(L)}} = \frac{\partial C}{\partial a_j^{(L)}} \cdot \sigma'(z_j^{(L)})$ (1)

- Layer l: $\Delta_j^{(l)} = \frac{\partial C}{\partial z_j^{(l)}} = \frac{\partial C}{\partial b_j^{(l)}} \frac{\partial b_j^{(l)}}{\partial z_j^{(l)}} = \frac{\partial C}{\partial b_j^{(l)}}$ (2)

- Kettenregel: $\Delta_j^{(l)} = \frac{\partial C}{\partial z_j^{(l)}} = \sum_k \frac{\partial C}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}}$
 $= \sum_k \Delta_k^{l+1} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = \sum_k \Delta_k^{l+1} \frac{\partial z_k^{(l+1)}}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}}$
- $= \left(\sum_k \Delta_k^{l+1} W_{kj}^{(l+1)} \right) f'(z_j^{(l)})$ (3)

- $\frac{\partial C}{\partial W_{jk}^{(l)}} = \frac{\partial C}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial W_{jk}^{(l)}} = \Delta_j^{(l)} a_k^{(l-1)}$ (4)

Notation:

$W_{jk}^{(l)}$: Gewicht, das Neuron k der Layer l - 1 und Neuron j der Layer l verbindet

$b_j^{(l)}$: Bias des Neurons j der Layer l

$a_j^{(l)}$: Aktivierung Neuron j der Layer l

$$z_j^{(l)} = \sum_k W_{jk}^{(l)} a_k^{(l-1)} + b_j^{(l)}$$

$\Delta_j^{(l)}$ Fehler des Neurons j der Layer l

$\hat{=}$ Änderung der Kostenfunktion C bzgl. $z_j^{(l)}$

$f(\cdot)$ nichtlineare Aktivierungsfunktionen

$$a_j^{(l)} = f(z_j^{(l)})$$

$\sigma(\cdot)$ (softmax) Aktivierungsfunktion der Neuronen der Output-Layer

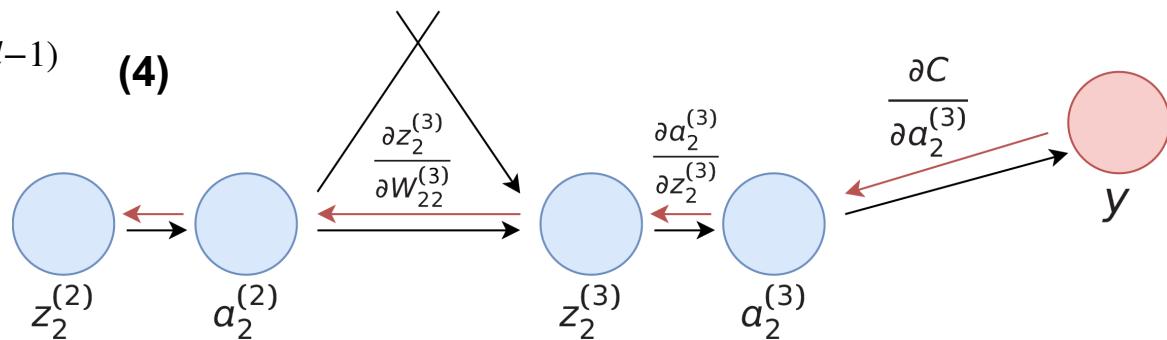


Illustration error Back-propagation

Back-propagation Algorithmus

$$\Delta_j^{(L)} = \frac{\partial C}{\partial a_j^{(L)}} \cdot \sigma' \left(z_j^{(L)} \right) \quad (1)$$

$$\Delta_j^{(l)} = \frac{\partial C}{\partial b_j^{(l)}} \quad (2)$$

$$\Delta_j^{(l)} = \left(\sum_k \Delta_k^{(l+1)} W_{kj}^{(l+1)} \right) f' \left(z_j^{(l)} \right) \quad (3)$$

$$\frac{\partial C}{\partial W_{jk}^{(l)}} = \Delta_j^{(l)} a_k^{(l-1)} \quad (4)$$

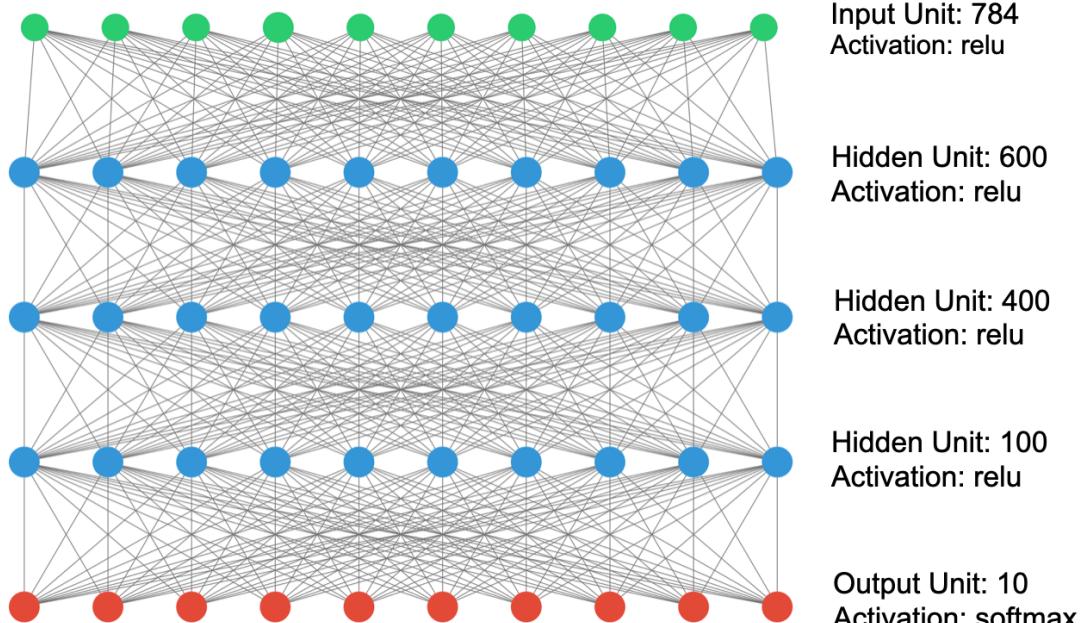
Algorithmus 2 Back-propagation Algorithmus

- 1: **Feedforward:** Berechne $\mathbf{a}^{(l)}$ und $\mathbf{z}^{(l)}$ für alle Schichten mit dem **Feedforward-Algorithmus**.
 - 2: **Fehler der Output-Layer:** Berechne mit (1) den Fehlerterm $\Delta_j^{(L)}$ für alle Neuronen der Output-Layer.
 - 3: **for** $l = L - 1, \dots, 1$ **do**
 - Berechne mit (3) den Fehlerterm $\Delta_j^{(l)}$ für alle Neuronen der l -ten Schicht.
 - Berechne mit (2) und (4) die Gradienten $\frac{\partial C}{\partial W_{jk}^{(l)}}$ und $\frac{\partial C}{\partial b_j^{(l)}}$.
 - 4: **end for**
-

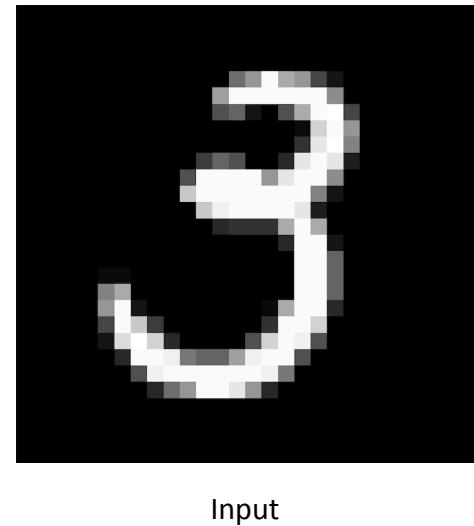
Mit dem Backpropagation Algorithmus kann man alle Gradienten berechnen, die für die Anpassung der Modellparameter mit Gradient-Descent benötigt werden

Klassifizierungsproblem MNIST Handwritten Digits

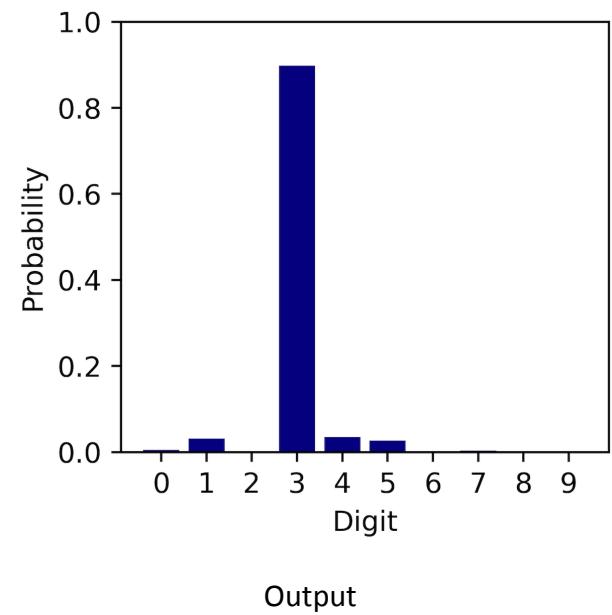
- Input: Bild einer Handgeschriebenen Ziffer als 28×28 Matrix mit Einträgen $\in [0,1]$
- Output: Wahrscheinlichkeitsschätzung für jede der 10 möglichen Klassen (Ziffern von 0 bis 9)



Schema des verwendeten Neuronal Netzes



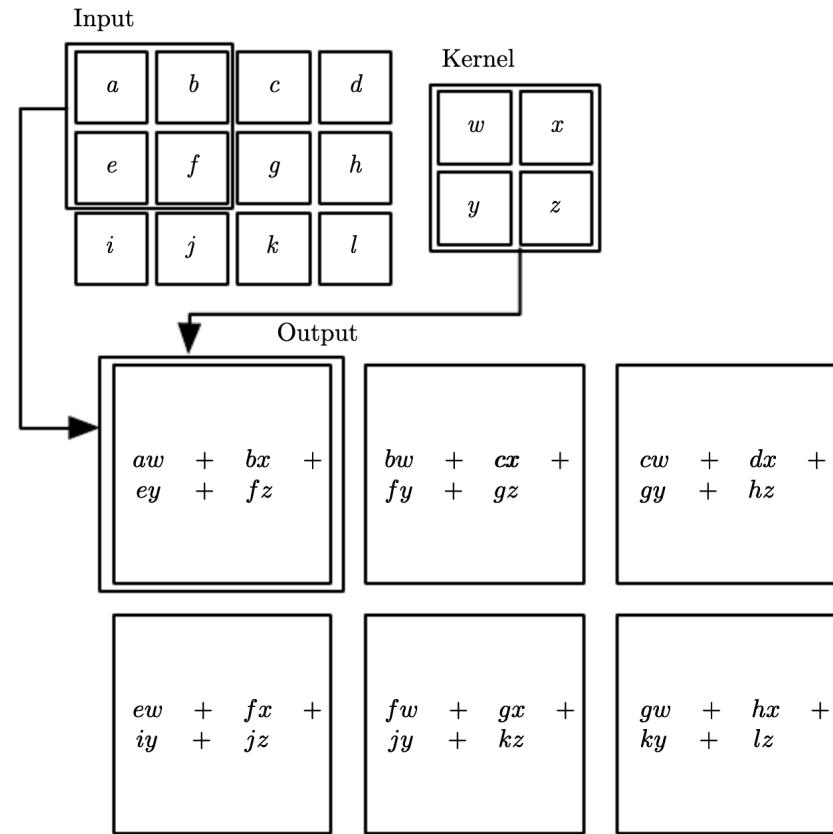
Input



Output

Convolutional Neural Networks

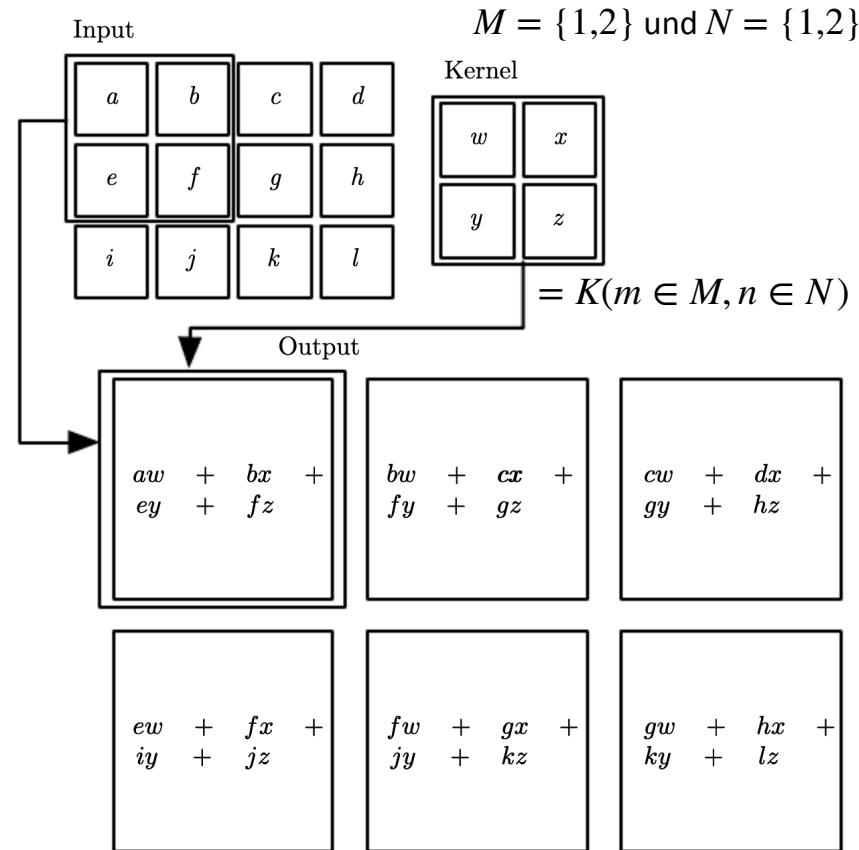
- Mehrere Layers möglich
- 1. Convolutional Layer:
 - Faltungsmatrizen auch (convolution) Kernel genannt, kleiner als Input
 - Mehrdimensionale Kernel in einer Layer möglich um verschiedene Merkmale herauszufiltern
 - Reduziert Speicheranforderungen, Rechenoperationen
 - Laufzeit für Schicht mit m inputs und n outputs:
Vollständig verbundene Schichten $O(m \times n)$, Kernel mit k einträgen $O(k \times n)$
 - Höhere statistische Effizienz, i.e. weniger Trainingsdaten zum lernen benötigt als bei vollständig verbundenen Netze



Goodfellow et al. Deep Learning 2016

Convolutional Neural Networks

- 2D Faltung z.B. Bild als Input:
- $S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(i, j)K(i - m, j - n)$
- $S(i, j) = (K * I)(i, j) = \sum_m \sum_n I(i - m, j - n)K(m, n)$
- Faltung Kommutativ
- Kreuz-Korrelation, nicht kommutativ:
- $S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(i + m, j + n)K(m, n)$
- Implementierung des Kernels: 4D Tensor $K_{i,l,m,n}$ mit folgenden Indices:
 - Index l für Input Kanal, z.B. RGB Werte bei Bildern
 - Index m, n für räumliche Koordinaten
 - Index i für Output Kanal
- Faltung von \mathbf{K} über Input \mathbf{V} : $Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m-1,k+n-1} K_{i,l,m,n}$



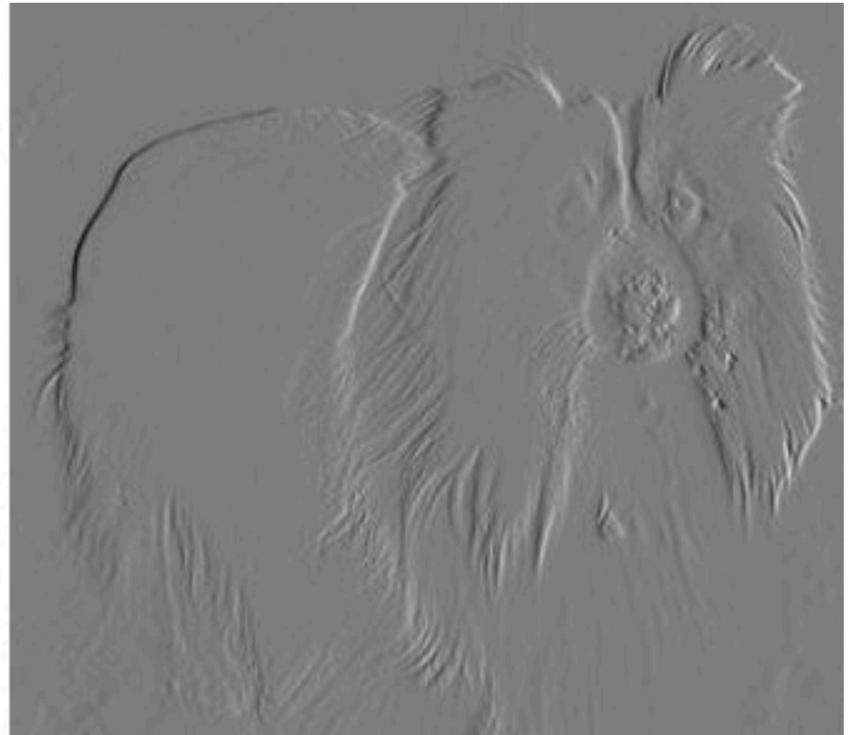
Goodfellow et al. Deep Learning 2016

$$S(1,1) = \sum_{m=1}^2 \sum_{n=1}^2 I(1 + m, 1 + n)K(m, n)$$

Convolutional Neural Networks: Beispiel

$$S(i, j) = \sum_m \sum_n I(i + m, j + n)K(m, n)$$

Hier: $K(m) = (1, -1)$



Goodfellow et al. Deep Learning 2016

Convolutional Neural Networks

- Erweiterungen möglich:
- **Strided Convolution:** Downsampling

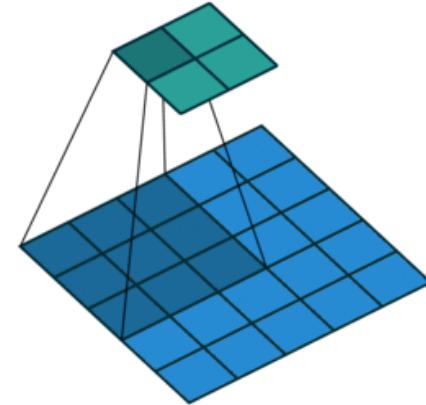
Möglich, z.B. nur alle s Pixel in jede Richtung in der Ausgabe abtasten:

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,(j-1)s+m,(k-1)s+n} K_{i,l,m,n}$$

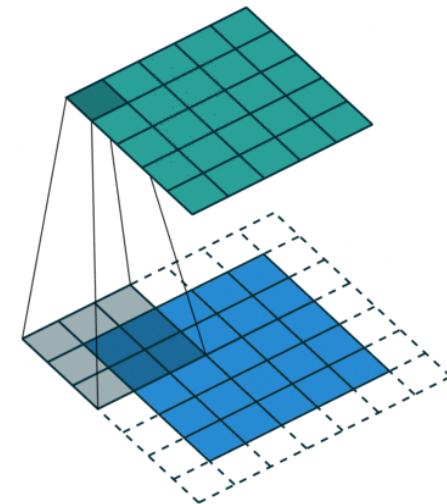
- **Tilted Convolution:** Eine Menge von Kerneln wechselt rotierend, während der Bewegung durch den Raum. Dadurch erhalten benachbarte Orte unterschiedliche Filter:

$$Z_{i,j,k} = \sum_m \sum_n V_{l,j+m-1,k+n-1} K_{i,l,m,n, j \% t + 1, k \% t + 1}$$

- **Padding:** Rand aus Nullen an zu den Inputdaten hinzufügen, um Größe des Inputs beizubehalten



Strided Convolution [1]



Padding [1]

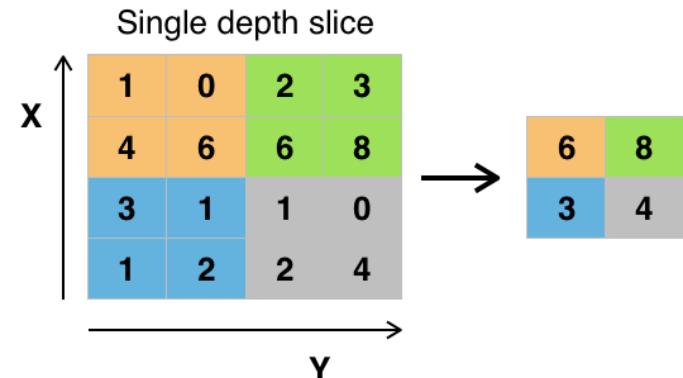
Convolutional Neural Networks

2. Detector Layer:

- Wendet Aktivierungsfunktion auf output der Convolutional Layer an

3. Pooling Layer:

- Ersetzt die Ausgabe einer Layer durch zusammenfassende Statistik benachbarter outputs
- Oft Max-Pooling: maximaler Wert innerhalb einer Umgebung eines outputs
- Ähnlich zu Lateraler Hemmung in der Neurobiologie



Max-Pooling https://commons.wikimedia.org/wiki/File:Max_pooling.png 2015

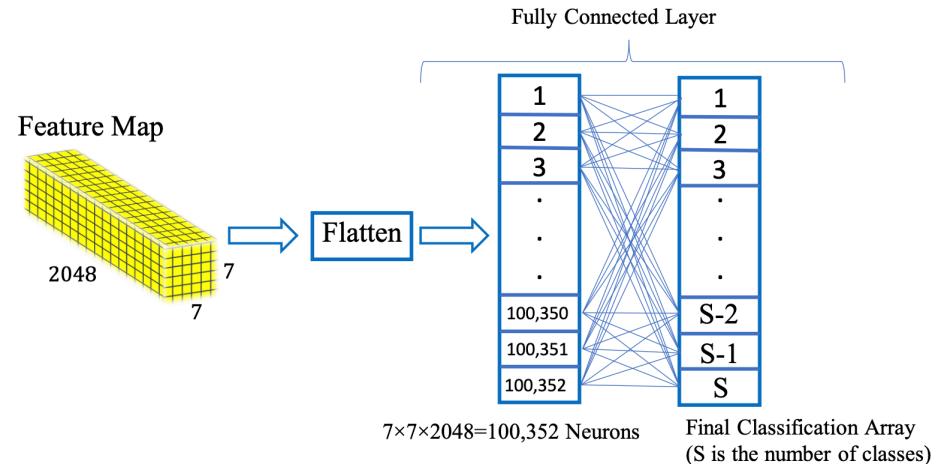
4. Dropout Layer

- Bei jedem Trainingsdurchlauf wird der Wert einzelner Knoten im Netzwerk mit gewisser Wahrscheinlichkeit P_{dropout} auf 0 gesetzt
- Dadurch bei jedem Trainingsdurchlauf leicht geänderte Netzwerkarchitektur
- Bessere Generalisierung des Modells, Maßnahme gegen overfitting

Convolutional Neural Networks

5. Flattening Layer

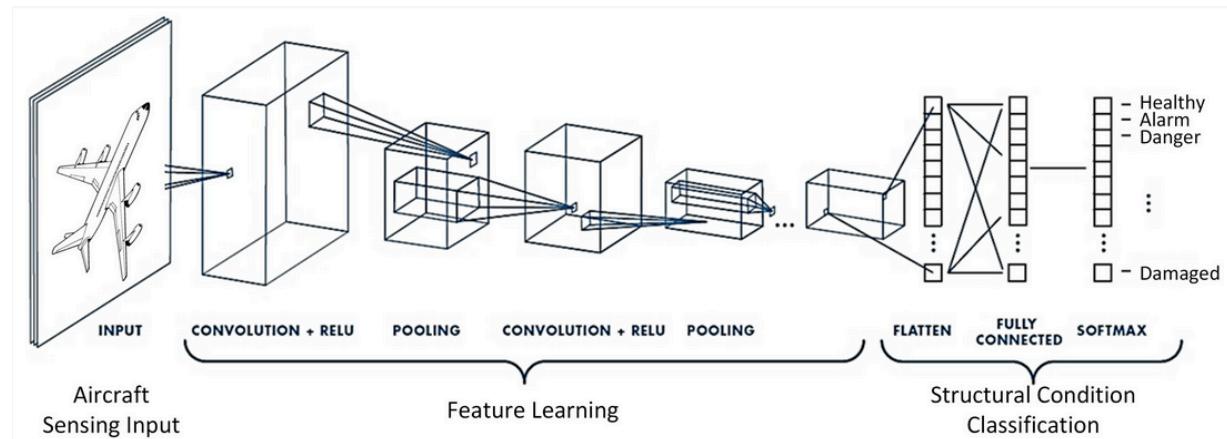
- Wandelt output in input für fully Layers um
- z.B. Klassifizierung des Outputs



- Convolutional Neural Networks bestehen aus vielen Layers der verschiedenen Typen

M. Rahimzadeh et al. Research Gate 2021

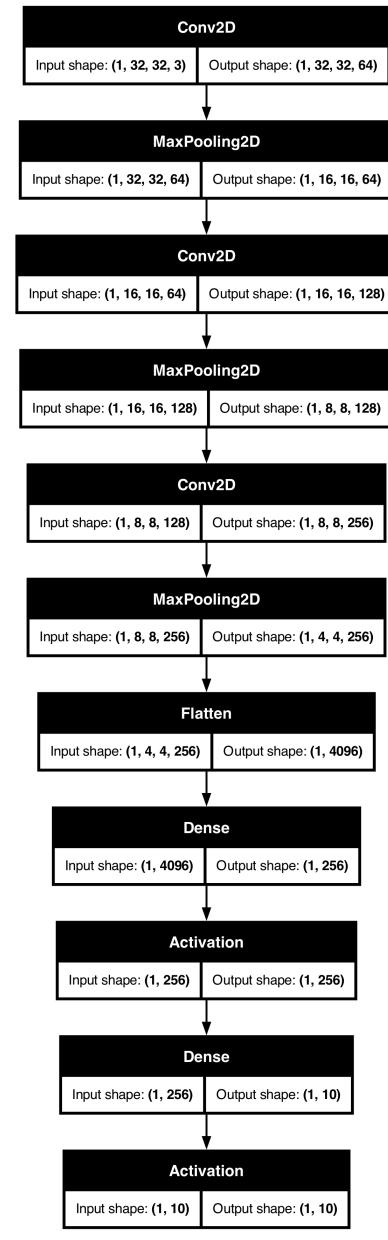
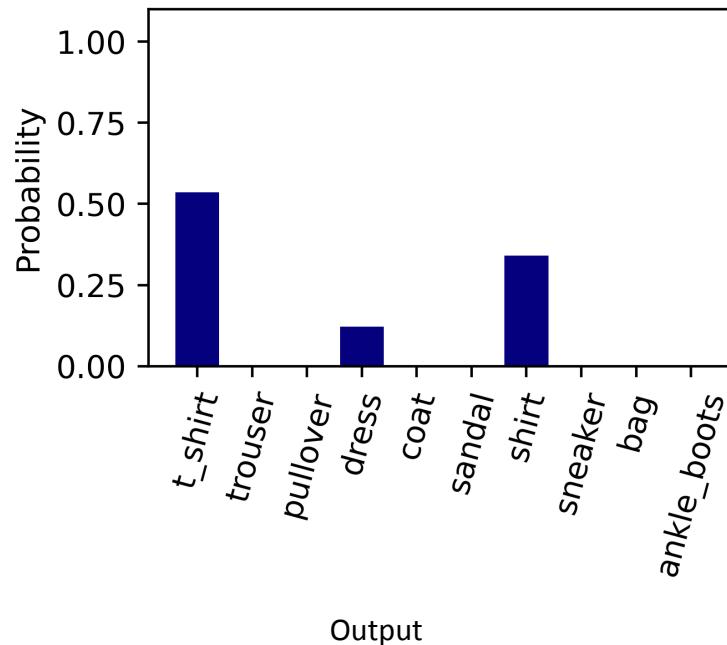
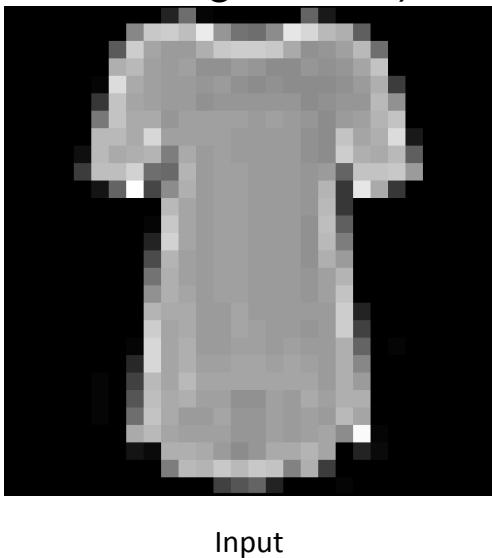
- Learning mittels Backpropagation



Typischer Aufbau eines Convolutional Neural Network. Iuliana Tabian et al. Convolutional Neural Network for Impact Detection and Characterization of Complex Composite Structures. *Sensors* 2019

Klassifizierungsproblem MNIST Fashion

- Input: Bild eines Kleidungsstücks als 28×28 Matrix mit Einträgen $\in [0,1]$
- Output: Wahrscheinlichkeitsschätzung für jede der 10 möglichen Klassen (verschiedene Kleidungsstücke)



Aufbau Netzwerks

Literaturverzeichnis

1. Ian Goodfellow and Yoshua Bengio and Aaron Courville. *Deep Learning*. MIT Press. 2016. <http://www.deeplearningbook.org>
2. M. Jordan and J. Kleinberg and B. Schölkopf. *Pattern Recognition and Machine Learning*. Springer Science+Business Media, LLC. 2006.
doi: 10.1007/978-0-387-45528-0
3. Pankaj Mehta and Marin Bukov and Ching-Hao Wang and Alexandre G.R. Day and Clint Richardson and Charles K. Fisher and David J. Schwab. *A high-bias, low-variance introduction to Machine Learning for physicists*. *Physics Reports*. 2019. Vol. 810. doi: 10.1016/j.physrep.2019.03.001
4. Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press. 2015.
5. Brady Neal. *On the Bias-Variance Tradeoff: Textbooks Need an Update*. arXiv. 2019. doi: 1912.08286