Lecture 4: Neural Networks and Backpropagation

BA-INF 153: Einführung in Deep Learning für Visual Computing

Prof. Dr. Reinhard Klein

Nils Wandel

Informatik II, Universität Bonn

08.05.2024

Recap

- Fehlerfunktionen
- Gradient Descent
- Stochastic Gradient Descent
- Adaptive Learning Rates

Today's Lecture

Themen für heute

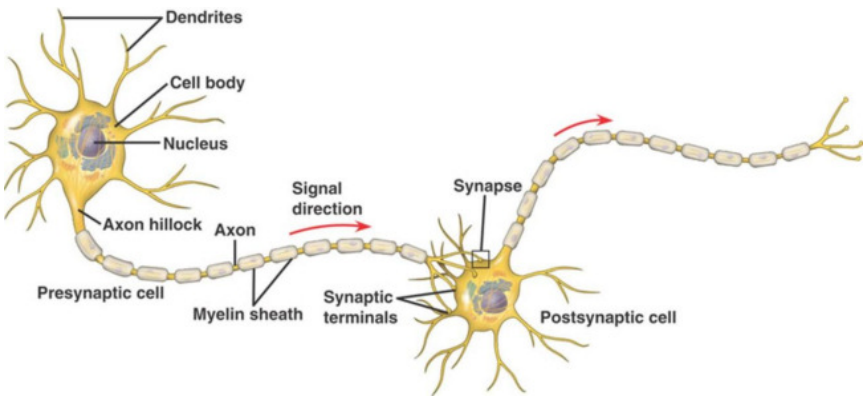
1. Neuronale Netze
2. Aktivierungsfunktionen
3. Backpropagation-Algorithmus

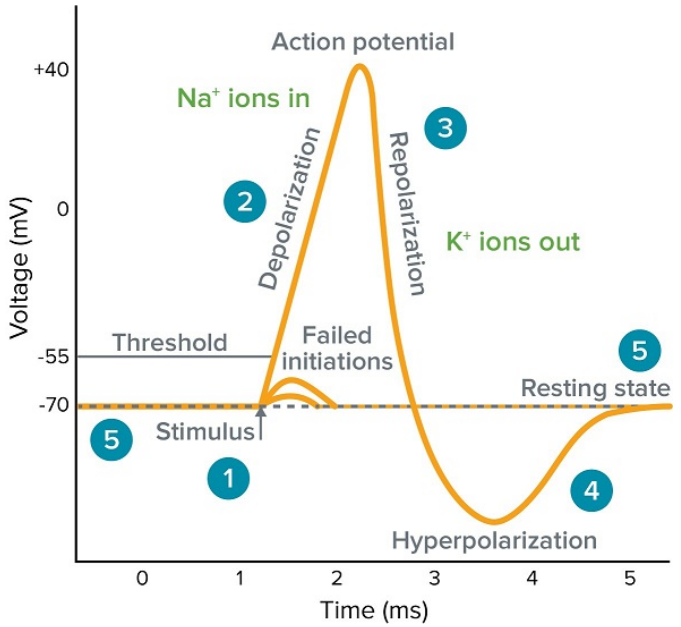
Part 1

Feed-Forward Networks

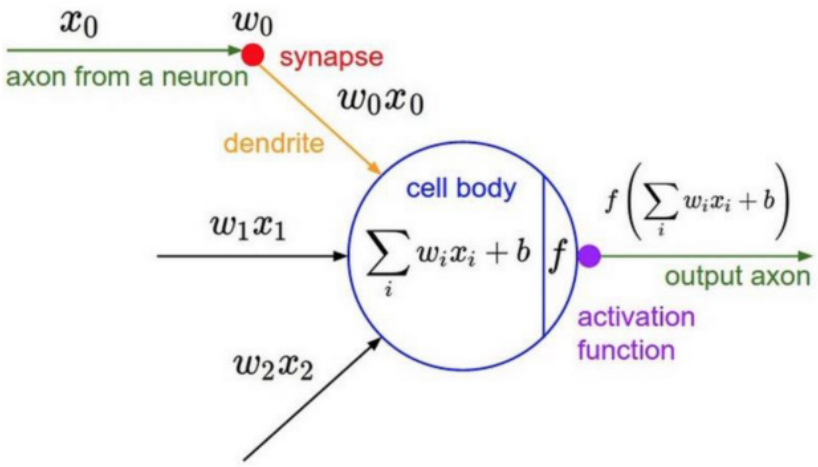
Chapter 5.1 of **bishop2006pattern**

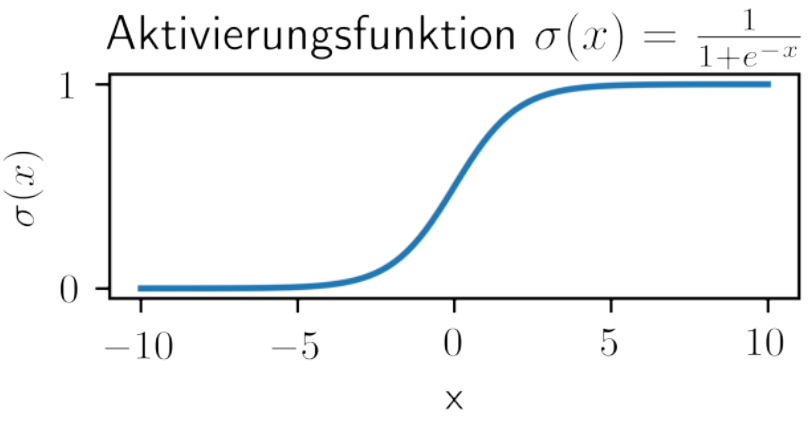
Künstliche neuronale Netze - Reminder





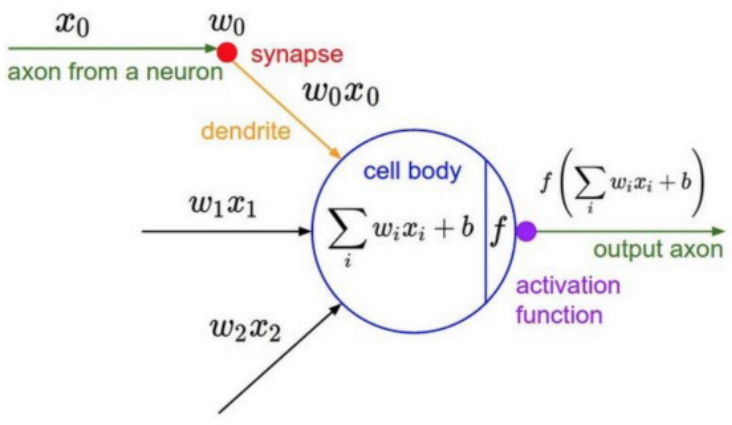
Mathematisches Modell und Beispiel für Activation Function (Sigmoid)

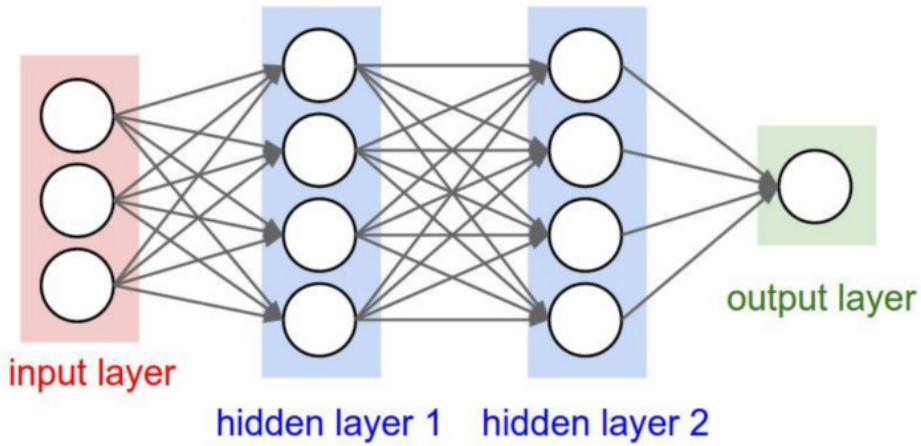




Künstliche neuronale Netze - Reminder

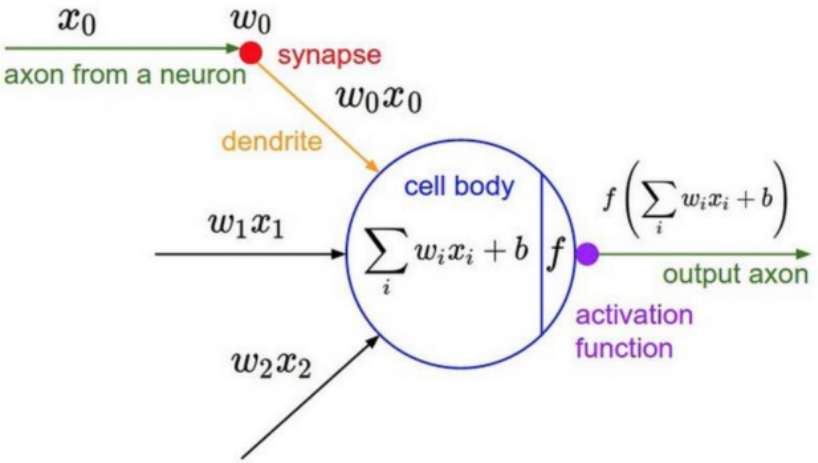
Einfaches Multilayer Perceptron (MLP)

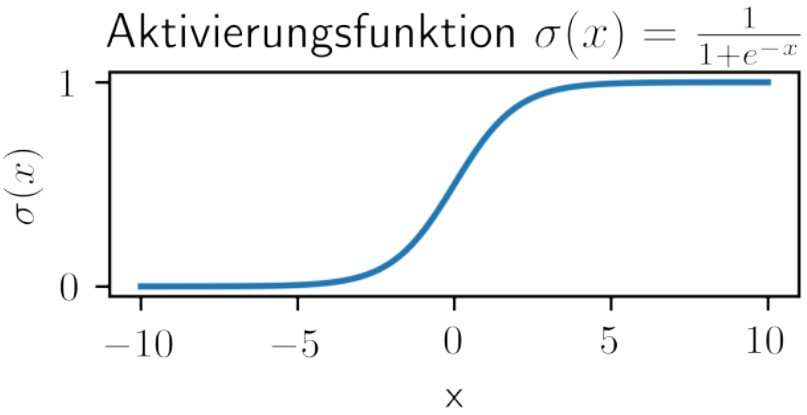




$\Rightarrow$ Ein Neuronales Netz entspricht einer parametrisierten Funktion, welche Eingabedaten auf Ausgabewerte abbildet.

Definition - einzelnes Neuron





Die Ausgabe ("Feuerrate") eines einzelnen Neurons mit D Eingabewerten x_i und synaptischen Gewichten w_i sowie Bias b kann wie folgt berechnet werden: Zunächst berechnet man die Aktivierung ("Membranpotential"):

$$\underbrace{a}_{\text{activation}} = \sum_{i=1}^{D} w_i \cdot x_i + \underbrace{b}_{\text{bias}}, \qquad j = 1, ..., M.$$

Danach wendet man eine *Aktivierungsfunktion* $f(\cdot)$ an, um die Ausgabe $z = f(a)$ des Neurons zu berechnen. Diese Aktivierungsfunktion sollte *nicht-linear* und *differenzierbar* sein (z.B. eine σ-Funktion), doch mehr dazu später...

Definition - einzelnes Layer ≈ mehrere Neuronen

Üblicherweise werden mehrere Neuronen auf die selben Eingaben angewendet und in einem Layer zusammengefasst. Die Ausgabe eines solchen Layers mit M Neuronen und D Eingabewerten kann wie folgt berechnet werden:

$$\underbrace{a_j}_{\text{activation}} = \sum_{i=1}^{D} w_{ji} \cdot x_i + \underbrace{b_j}_{\text{bias}}, \qquad j = 1, ..., M.$$

Danach wendet man wieder eine Aktivierungsfunktion $f(\cdot)$ auf jeden Aktivierungswert an, um die Ausgaben der Neuronen des Layers zu berechnen:

$$z_j = f(a_j)$$

Etwas kompakter lässt sich dies mit Hilfe von Vektoren und Matrizen aufschreiben:

$$\boldsymbol{a} = W\boldsymbol{x} + \boldsymbol{b}$$

$$\boldsymbol{z} = f(\boldsymbol{a})$$

Wobei $W \in \mathbb{R}^{M \times D}$ eine Matrix ist, welche die synaptischen Gewichte w_{ji} enthält und $\boldsymbol{b} \in \mathbb{R}^M$ ein Vektor ist, welcher die Bias-werte b_i enthält.

Definition - einzelnes Layer $\approx$ mehrere Neuronen

Bemerkung

In der Literatur wird die Gewichtsmatrix und der Biasvektor manchmal wie folgt zusammengeführt:

$$\boldsymbol{a} = [\boldsymbol{b}, W] \begin{bmatrix} 1 \\ \boldsymbol{x} \end{bmatrix} = \hat{W}\hat{\boldsymbol{x}}$$

... diese Notation erlaubt noch eine etwas kompaktere Darstellung. In dieser Vorlesung verwenden wir der Einfachheit halber trotzdem eine getrennte Darstellung für W und $\boldsymbol{b}$.

Definition - Neuronales Netz ≈ mehrere Layers

Ein neuronales Netz enthält üblicherweise eine Vielzahl an Layern (hier: N) welche nacheinander angewendet werden:

$$\begin{aligned}
\boldsymbol{a}^{(1)} &= W^{(1)} \cdot \boldsymbol{x} + \boldsymbol{b}^{(1)}, & \boldsymbol{z}^{(1)} &= f^{(1)}(\boldsymbol{a}^{(1)}) \\
\boldsymbol{a}^{(2)} &= W^{(2)} \cdot \boldsymbol{z}^{(1)} + \boldsymbol{b}^{(2)}, & \boldsymbol{z}^{(2)} &= f^{(2)}(\boldsymbol{a}^{(2)}) \\
&\vdots & &\vdots \\
\boldsymbol{a}^{(N)} &= W^{(N)} \cdot \boldsymbol{z}^{(N-1)} + \boldsymbol{b}^{(N)}, & \boldsymbol{y} &= f^{\text{out}}(a^{(N)})
\end{aligned}$$

Hierbei sind $\boldsymbol{x}$ der Inputvektor, $W^{(i)}$, $\boldsymbol{b}^{(i)}$, $\boldsymbol{a}^{(i)}$, $f^{(i)}(\cdot)$ und $z^{(i)}$ die Gewichtsmatrizen, Biasvektoren, Aktivierungswerte, Aktivierungsfunktionen und Ausgaben des i-ten Layers. f^{out} und $\boldsymbol{y}$ entsprechen der Aktivierungsfunktion des Ausgabelayers respektive der Ausgabe des neuronalen Netzes.

Definition - Neuronales Netz ≈ mehrere Layers

Man kann das Neuronale Netz nun auch als eine parametrisierte Funktion auffassen:

$$nn_\theta(\boldsymbol{x}) = f^{\text{out}}(W^{(N)} \cdot f^{(N-1)}(...W^{(2)} \cdot f^{(1)}(W^{(1)} \cdot \boldsymbol{x} + \boldsymbol{b}^{(1)}) + \boldsymbol{b}^{(2)}...) + \boldsymbol{b}^{(N)})$$

Die Parameter entsprechen hier: $\theta = ((W^{(1)}, \boldsymbol{b}^{(1)}), (W^{(2)}, \boldsymbol{b}^{(2)}), ..., (W^{(N)}, \boldsymbol{b}^{(N)}))$

Wie viele Layer hat ein Neuronales Netz?

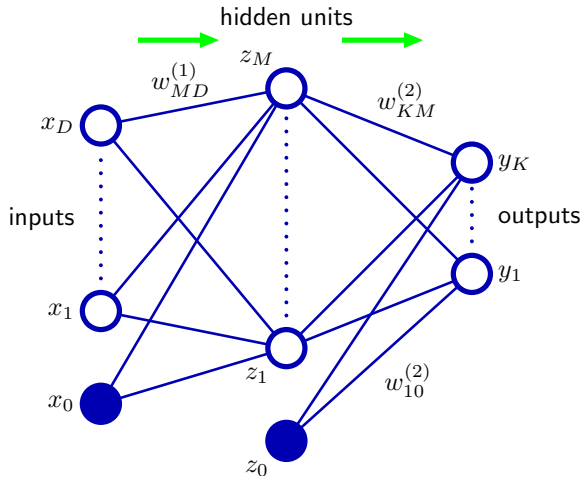


Figure: 5.1 von Bishop

Was macht ein "Layer" aus? Die Terminologie in der Literatur ist nicht immer konsistent: Das obige Netzwerk kann als 3-Layer Netzwerk betrachtet werden (Eingabe + hidden Layer + Ausgabe) oder als single-hidden-layer Netzwerk (Neurone in einem hidden Layer werden auch "hidden units" genannt). Bishop empfielt, dieses Netzwerk als 2-Layer Netzwerk zu bezeichnen, wobei die Anzahl Layer der Anzahl der Multiplikationen mit Gewichtsmatrizen $W^{(i)}$ entspricht.

Part 2

Aktivierungsfunktionen

Chapter 6.2 of **Goodfellow-et-al-2016-Book**

Lineare vs. Nicht-Lineare Aktivierungsfunktionen

Warum benötigen wir *nicht-lineare* Aktivierungsfunktionen?

Lineare Abbildungen

Eine lineare Abbildung $f(x)$ hat folgende Eigenschaften:

- *Additivität* $f(x+y) = f(x) + f(y)$
- *Homogenität* $f(\alpha x) = \alpha f(x)$.

Lineare vs. Nicht-Lineare Aktivierungsfunktionen

Nehmen wir an, wir hätten ein Netzwerk mit N Layers ohne Bias-Terme, wobei die Aktivierungsfunktionen $f(\cdot)$ linear sind:

$$
\begin{aligned}
nn_\theta(x) &= f^{\text{out}}\Big(W^{(N)} \cdot f^{(N-1)} \dots f^{(2)}\big(W^{(2)} \cdot f^{(1)}(W^{(1)} \cdot \boldsymbol{x})\big)\Big) \\
&= f^{\text{out}}\Big(W^{(N)} \cdot f^{(N-1)} \dots f^{(2)}\big(W^{(2)} \cdot \underbrace{W^{(1)} \cdot f^{(1)}}_{\substack{\text{weighting outside} \\ \text{of activation}}}(\boldsymbol{x})\big)\Big) \\
&= f^{\text{out}}\Big(\underbrace{W^{(N)} \dots W^{(2)} \cdot W^{(1)}}_{W'} \cdot \underbrace{f^{(N-1)} \circ f^{(2)} \circ f^{(1)}}_{g}(\boldsymbol{x})\Big) \\
&= f^{\text{out}}\big(W' g(\boldsymbol{x})\big)
\end{aligned}
$$

Dies zeigt uns, dass ein Multi-Layer Netzwerk mit linearen Aktivierungsfunktionen auch durch ein äquivalentes kollabiertes Netzwerk beschrieben werden kann. Ein Multi-Layer Linear Network hat also nicht mehr Aussagekraft als ein Single-Layer Linear Network und ist somit uninteressant für uns. Hier haben wir angenommen, dass die Bias-Terme $\boldsymbol{b}^{(i)} = 0$ sind. Man kann jedoch mit Hilfe der Additivität zeigen, dass Netzwerke mit Bias-Termen ebenfalls kollabieren.

Aktivierungsfunktionen - Übersicht

Hidden units

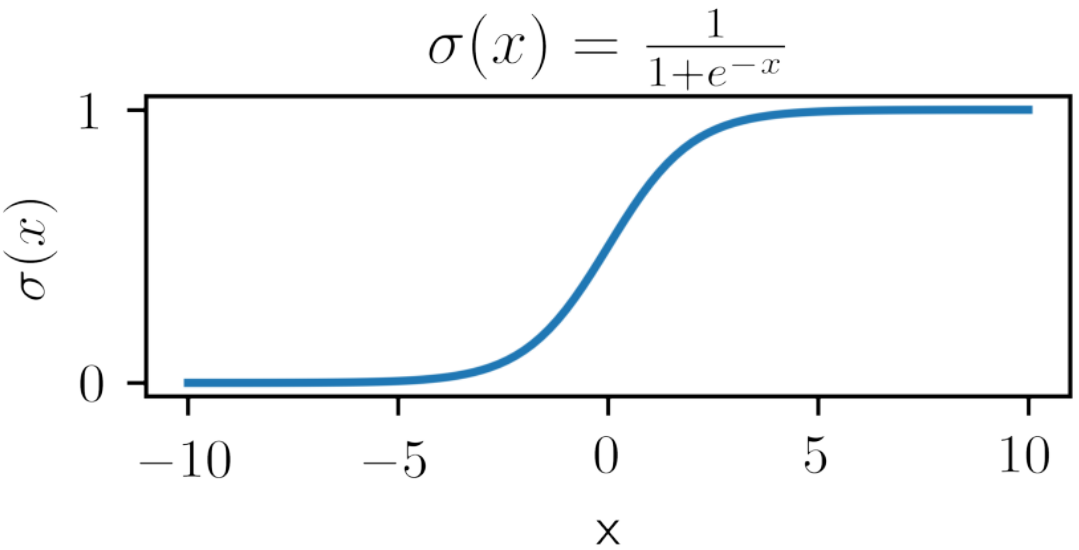
- sigmoid
- tanh
- ReLU
- LeakyReLU
- Parametric ReLU
- Softplus
- GELU
- Sinus / Cosinus

Output units

- Linear Unit (Identität)
- Softmax

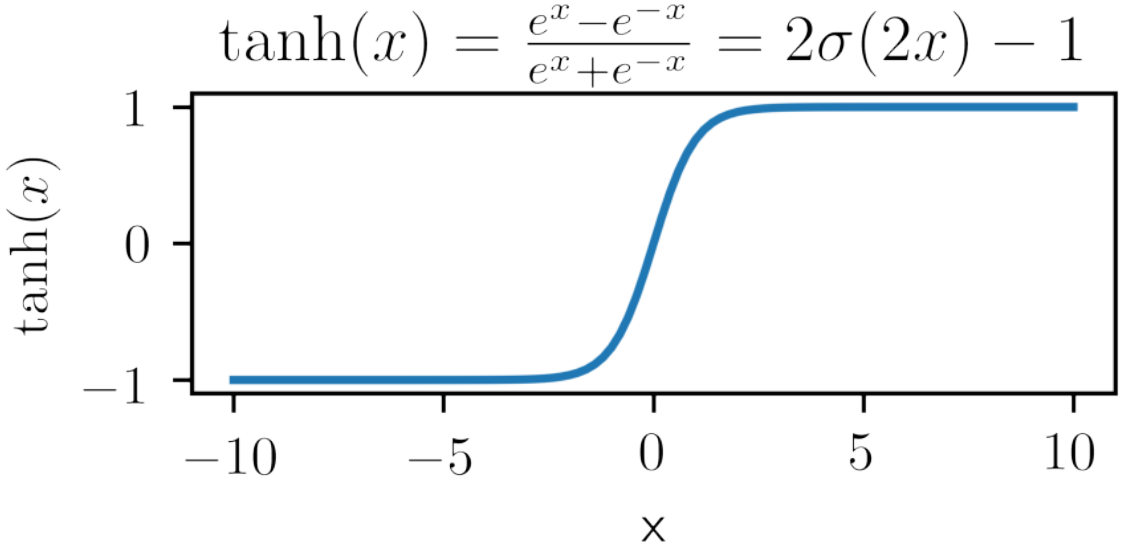
Aktivierungsfunktionen für Hidden Units

Sigmoid



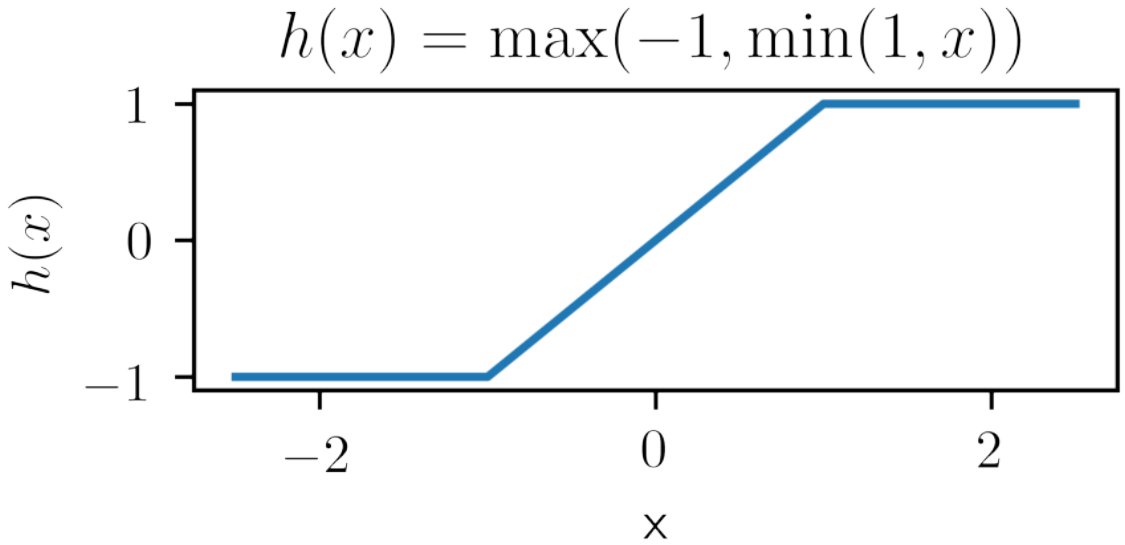
Aktivierungsfunktionen für Hidden Units

Tangens Hyperbolicus (tanh)



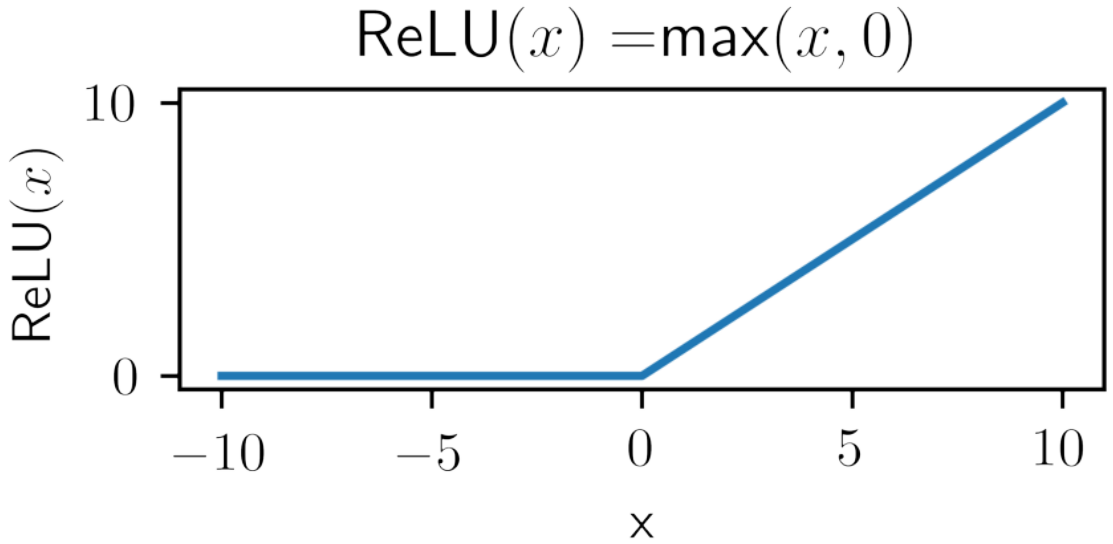
Aktivierungsfunktionen für Hidden Units

Hard tanh



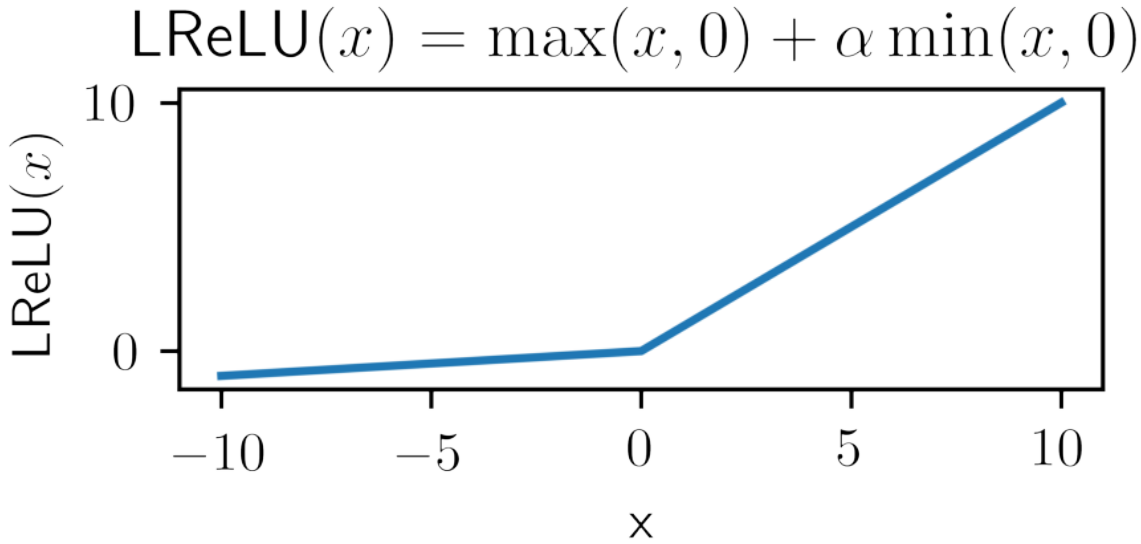
Aktivierungsfunktionen für Hidden Units

Rectified Linear Unit (ReLU)



Aktivierungsfunktionen für Hidden Units

Leaky ReLU



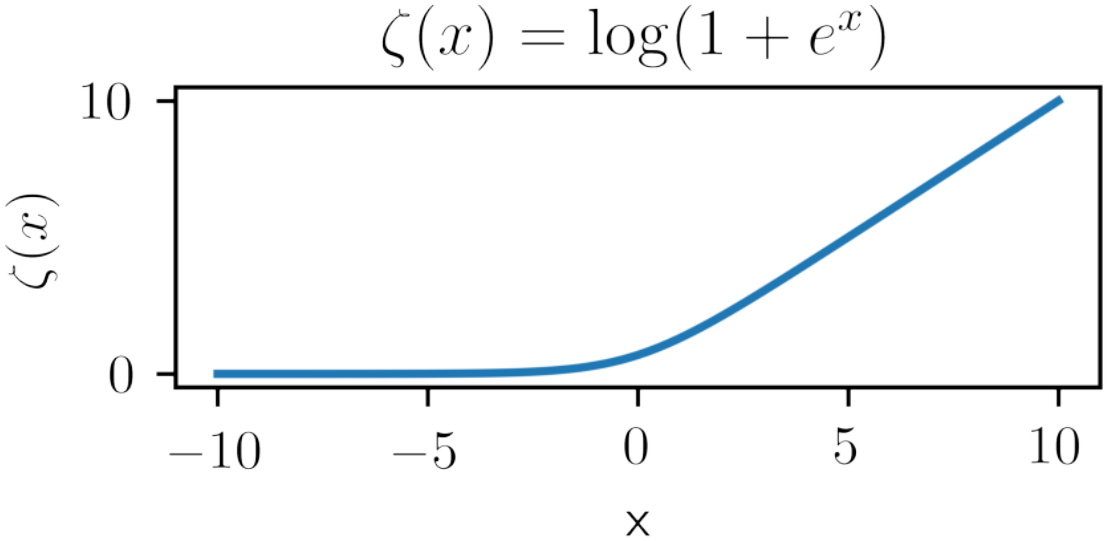
Typischer Wert für $\alpha = 0.01$.

Parametric ReLU

Ähnlich wie Leaky ReLU, allerdings wird α als lernbarer Parameter behandelt.

Aktivierungsfunktionen für Hidden Units

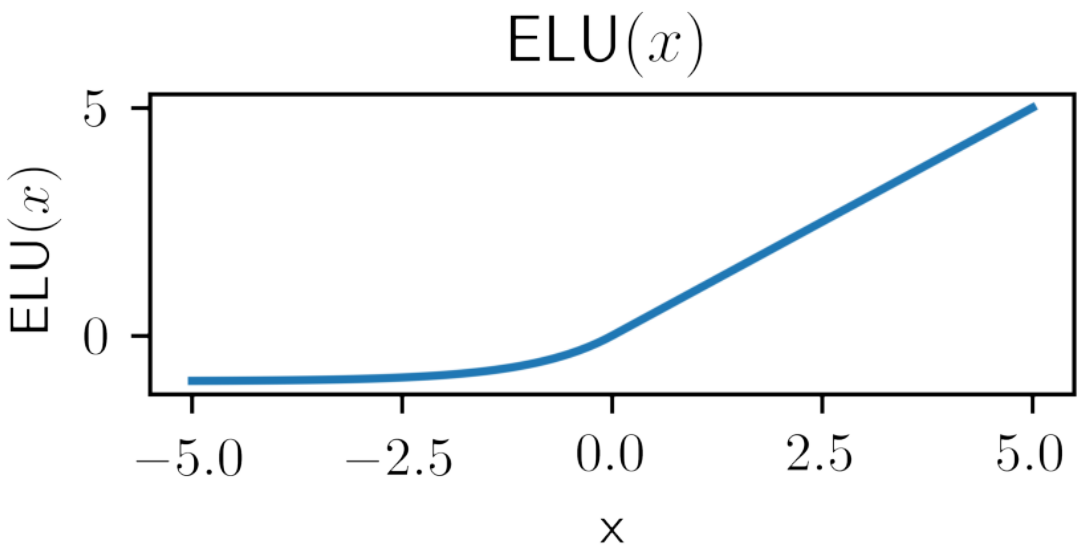
Softplus



Aktivierungsfunktionen für Hidden Units

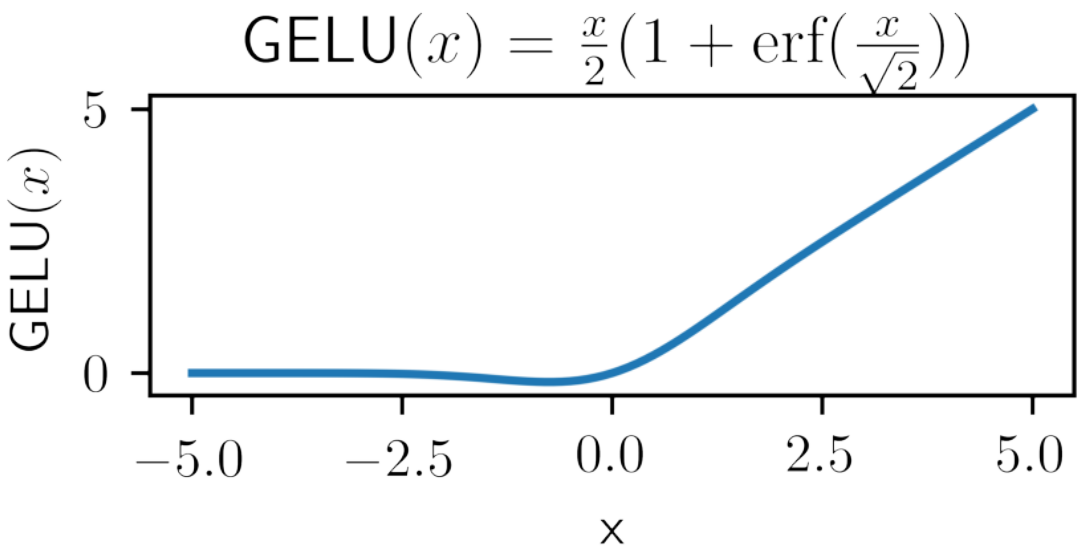
Exponential Linear Unit (ELU)

$$\text{ELU}(x) = \begin{cases} x & \text{wenn x} \geq 0 \\ e^x - 1 & \text{wenn x} < 0 \end{cases}$$



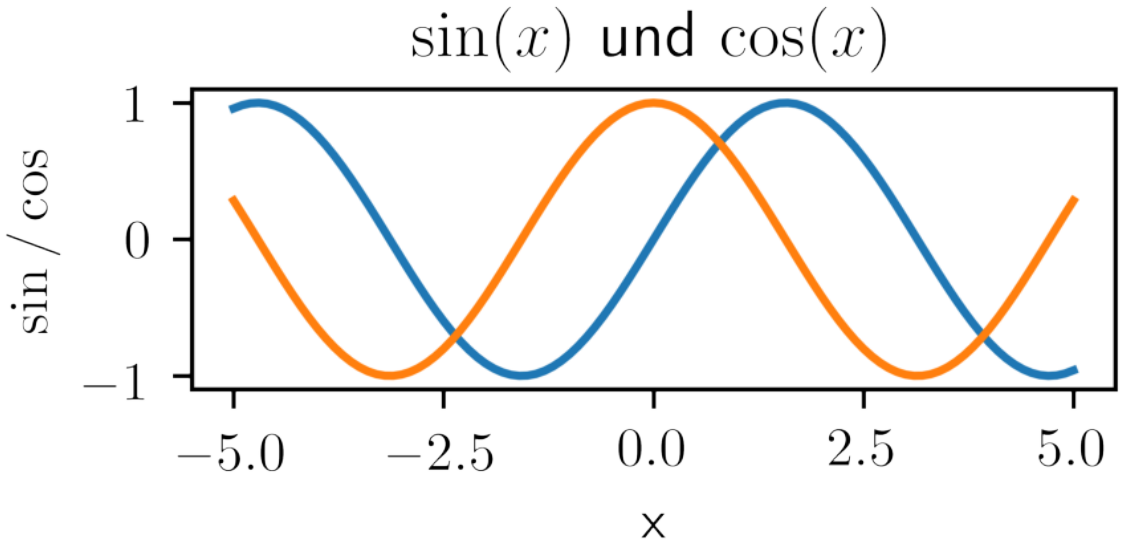
Aktivierungsfunktionen für Hidden Units

Gaussian Error Linear Unit (GELU)



Aktivierungsfunktionen für Hidden Units

Sinus / Cosinus



... weitere schöne Darstellungen von Aktivierungsfunktionen können [hier] gefunden werden.

Aktivierungsfunktionen für Output Units

Linear Unit

$$l(\boldsymbol{x}) = \boldsymbol{x}$$

... diese Abbildung entspricht der Identität und ist nützlich als Ausgabe für Regressionsprobleme (z.B. für den Mittelwert $\mu(x)$ einer bedingten Normal- oder Laplaceverteilung).
Beachte: wie bereits oben diskutiert, sollte eine solche lineare Funktion nicht in hidden Layers verwendet werden, da sich das Netzwerk sonst kollabieren lässt.

Aktivierungsfunktionen für Output Units

Softmax

$$\text{softmax}(\boldsymbol{x}) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Die Ausgabe von Softmax kann als Wahrscheinlichkeitsfunktion $p(y = i|\boldsymbol{x})$ aufgefasst werden und ist nützlich für Klassifikations- und Segmentierungsprobleme. Beachte, dass der Softmax invariant bezüglich der Addition eines Skalarwertes ist: $\text{softmax}(\boldsymbol{x}) = \text{softmax}(\boldsymbol{x} + c)$. Bei großen oder kleinen c können dadurch numerisch problematische Werte im Exponenten entstehen. Um dies zu verhindern, kann folgender numerisch stabilerer Softmax verwendet werden:

$$\text{softmax}_{\text{stable}}(\boldsymbol{x}) = \text{softmax}(\boldsymbol{x} - \max_i(\boldsymbol{x}))$$

Sigmoid

Die σ-Funktion kann zur binären Klassifikation oder Segmentierung verwendet werden. Sie ist sehr ähnlich zu Softmax, wenn man die Differenz zwischen x_1 und x_2 als Eingabe betrachtet.

Symmetrien im Parameter-Raum

Es gibt viele Möglichkeiten um die selben Abbildungen von Eingaben $\boldsymbol{x}$ auf Ausgaben $\boldsymbol{y}$ mit unterschiedlichen Parametern (Gewichts-Matrizen und Bias-Vektoren) zu erreichen.

Permutation der Neuronen

Wir können die Eingaben und Gewichte von 2 beliebigen Neuronen eines Layers vertauschen ohne das Ergebnis zu verändern. Bei M Neuronen führt dies zu $M!$ möglichen Permutationen.

Symmetrische Aktivierungsfunktionen

Betrachte ein 2-Layer Netzwerk mit $\tanh$ Aktivierungsfunktionen. Da $\tanh(-x) = -\tanh(x)$ ist, können wir aufeinander folgende Gewichte w_{ji} und w_{kj} mit -1 multiplizieren, ohne das Ergebnis zu verändern. Somit führen M Neuronen in einem Layer zu 2^M unterschiedlichen Möglichkeiten, das selbe Ergebnis zu bekommen. Wenn man den Bias-Vektor ebenfalls berücksichtigt, lässt sich diese Symmetrie auch auf weitere Aktivierungsfunktionen anwenden (z.B. $\sigma(-x) = 1 - \sigma(x)$).

Symmetrien und lokale Minima

Symmetrien

Kombiniert führen diese Symmetrien in einem Layer mit M Neuronen zu einem Faktor von $M! \cdot 2^M$ Symmetrien für den Parameterraum.
Aufgrund dieser Symmetrien in den Parametern gibt es viele lokale Minima (angenommen, man hätte ein Minimum gefunden, dann gäbe es für jedes Layer auf Grund der Symmetrien $M! \cdot 2^M$ weitere äquivalente Minima).

Globale Minima sind nicht nötig

Lokale Minima sind problematisch, wenn sie deutlich höhere Kosten als globale Minima haben. Wenn solche lokalen Minima vorkommen, kann dies ein Problem für gradientenbasierte Verfahren sein. Wenn neuronale Netze allerdings gross genug sind, dann haben lokale Minima üblicherweise bereits kleine Kosten, sodass es nicht so wichtig ist, ein globales Minimum zu finden.

Universal Approximation Theorem

Neuronale Netze haben eine starke Modell-Kapazität. Bei genügend hidden units kann sogar ein 2-Layer Netzwerk jede kontinuierliche Funktion auf einer kompakten Domäne mit beliebiger Genauigkeit uniform approximieren. Dieses Resultat gilt für zahlreiche Aktivierungsfunktionen.

Figure 5.3 Illustration of the capability of a multilayer perceptron to approximate four different functions comprising (a) $f(x) = x^2$, (b) $f(x) = \sin(x)$, (c), $f(x) = |x|$, and (d) $f(x) = H(x)$ where $H(x)$ is the Heaviside step function. In each case, $N = 50$ data points, shown as blue dots, have been sampled uniformly in x over the interval $(-1, 1)$ and the corresponding values of $f(x)$ evaluated. These data points are then used to train a two-layer network having 3 hidden units with 'tanh' activation functions and linear output units. The resulting network functions are shown by the red curves, and the outputs of the three hidden units are shown by the three dashed curves.

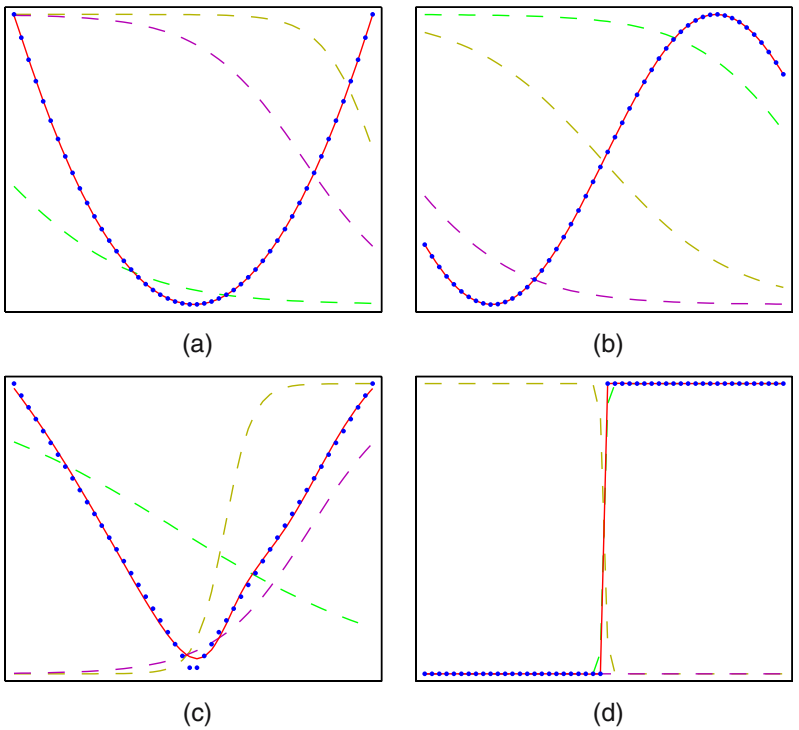


Figure: 5.3 von Bishop

Part 3

Backpropagation Algorithmus

Motivation: Backpropagation

In der letzten Vorlesung haben wir gesehen, dass gradientenbasierte Optimierungsverfahren wichtig sind, um den Fehler von Modellen mit vielen Parametern (z.B. neuronale Netzen) zu minimieren.
$\Rightarrow$ Wie können wir Gradienten einer Fehler-Funktion bezüglich der Parameter von neuronalen Netzen (also den Gewichts-Matrizen $W^{(i)}$ und Bias-Vektoren $\boldsymbol{b}^{(i)}$) effizient berechnen?

Motivation: Backpropagation

Ein naiver Ansatz wäre, Gradienten mit Hilfe von Differenzenquotienten zu berechnen:

$$\frac{\partial L}{\partial W_{ji}} = \frac{L(W_{ji} + \epsilon) - L(W_{ji} - \epsilon)}{2\epsilon} + \underbrace{O(\epsilon^2)}_{\text{residual corrections}} \qquad \text{wobei } \epsilon \ll 1.$$

Diese Art der numerische Differentiation ist bei einer hohen Anzahl von Parametern jedoch ineffizient, da wir für jeden Parameter die Loss-Funktion 2 Mal evaluieren müssen. Außerdem können die Residuen zu numerischen Ungenauigkeiten führen.
$\Rightarrow$ können wir die Gradienten effizienter und genauer berechnen?

Kettenregel in 1D

Der Backpropagation Algorithmus zur Berechnung von Ableitungen basiert auf der Kettenregel.
Nehmen wir als Beispiel die Funktion $g(f(x))$:

In 1D besagt die Kettenregel, dass:

$$(g \circ f)'(x) = g'(f(x))f'(x)$$

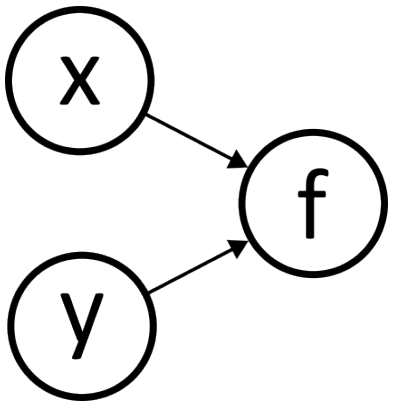
Manchmal wird die Kettenregel auch wie folgt dargestellt:

$$\frac{\partial}{\partial x} g(f(x)) = \frac{\partial g}{\partial f} \frac{\partial f}{\partial x}$$

Ableitung bei mehreren Eingaben

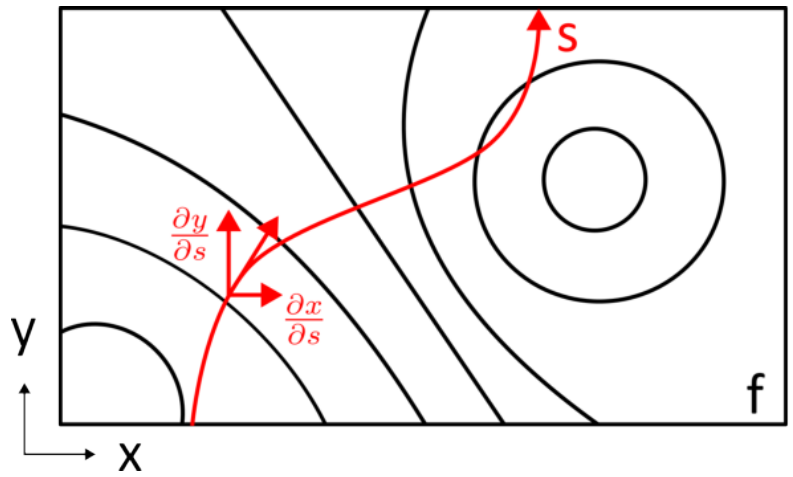
Was ist, wenn eine Funktion von mehreren Eingaben abhängt?

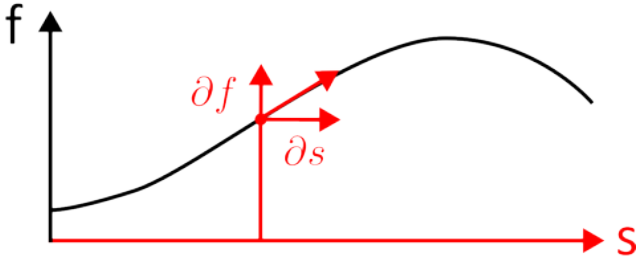
Wenn eine Funktion ($f(x, y)$) mehrere Eingaben (x, y) hat, dann können wir die partiellen Ableitungen $\frac{\partial f}{\partial x}$ bzw $\frac{\partial f}{\partial y}$ betrachten:



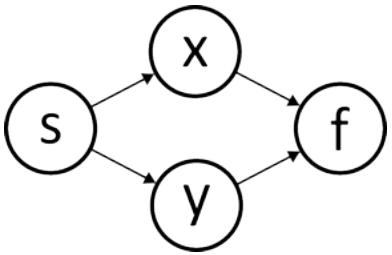
Richtungsableitung

Nehmen wir nun an, wir bewegen uns auf einem Pfad $(x(s), y(s))$:





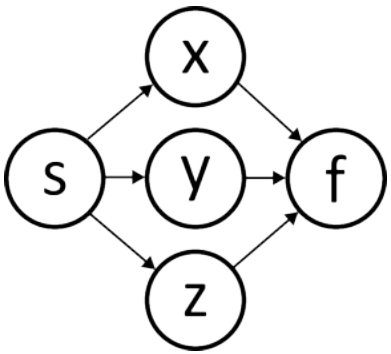
Um die partielle Ableitung von f nach s zu berechnen, müssen wir die Ableitung von f nach x und y, sowie den Einfluss von s auf x und y betrachten:



$$\frac{\partial f}{\partial s} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial s} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial s}$$

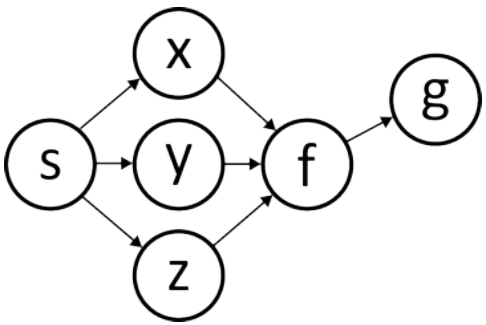
Im Berechnungsgraph müssen wir also beide Pfade, auf denen s Einfluss auf f nimmt aufaddieren.

Ableitungen im Berechnungsgraph - Beispiele



In 3D müsste auch die z Koordinate in der Berechnung der Ableitung berücksichtigt werden:

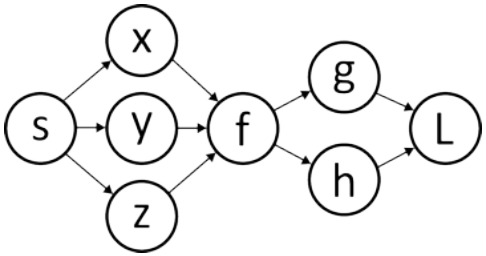
$$\frac{\partial f}{\partial s} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial s} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial s} + \frac{\partial f}{\partial z}\frac{\partial z}{\partial s}$$



Wenn wir nun die Ableitung nach einer weiteren Funktion g berechnen wollen, so müssen wir den Einfluss von f auf g, sowie den Einfluss von s auf f (siehe oben) betrachten:

$$\frac{\partial g}{\partial s} = \frac{\partial g}{\partial f}\frac{\partial f}{\partial s}$$

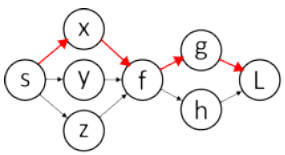
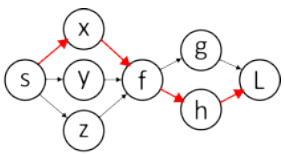
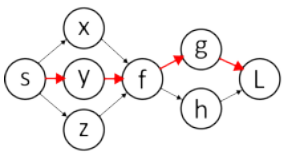
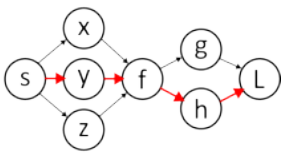
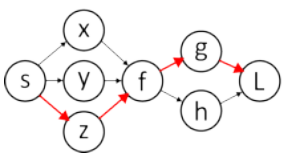
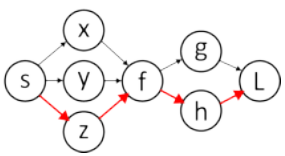
Ableitungen im Berechnungsgraph - Beispiele



Als Beispiel erweitern wir den Berechnungsgraphen um 2 weitere Funktionen (h und L):

$$\frac{\partial L}{\partial s} = \frac{\partial L}{\partial g}\frac{\partial g}{\partial s} + \frac{\partial L}{\partial h}\frac{\partial h}{\partial s}$$

Wenn wir nun alle Komponenten einsetzen und ausmultiplizieren, erhalten wir für jeden Pfad von s nach L einen Term:

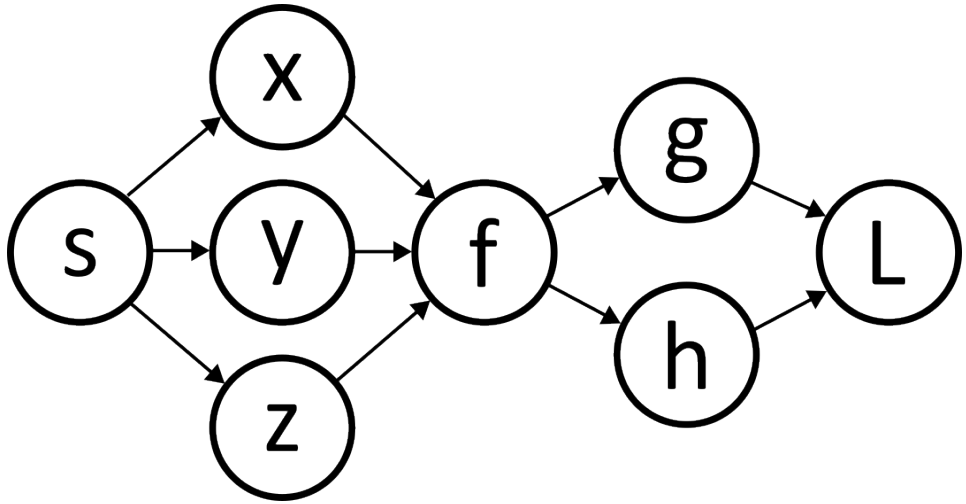
$$\frac{\partial L}{\partial s} = \frac{\partial L}{\partial g}\frac{\partial g}{\partial f}\frac{\partial f}{\partial x}\frac{\partial x}{\partial s} + \frac{\partial L}{\partial h}\frac{\partial h}{\partial f}\frac{\partial f}{\partial x}\frac{\partial x}{\partial s} + \frac{\partial L}{\partial g}\frac{\partial g}{\partial f}\frac{\partial f}{\partial y}\frac{\partial y}{\partial s} + \frac{\partial L}{\partial h}\frac{\partial h}{\partial f}\frac{\partial f}{\partial y}\frac{\partial y}{\partial s} + \frac{\partial L}{\partial g}\frac{\partial g}{\partial f}\frac{\partial f}{\partial z}\frac{\partial z}{\partial s} + \frac{\partial L}{\partial h}\frac{\partial h}{\partial f}\frac{\partial f}{\partial z}\frac{\partial z}{\partial s}$$

Ableitungen im Berechnungsgraph

Problem: Die Anzahl der Pfade von der Eingabe zur Ausgabe steigt exponentiell mit der Tiefe des Berechnungsgraphen. Für Große Netzwerke ist diese Art der Ableitungsberechnung somit unpraktikabel.
Wir können jedoch das Prinzip der dynamischen Programmierung verwenden, um die Ableitungen deutlich effizienter zu berechnen. Hierbei gibt es 2 Ansätze:

1. Wir fangen bei der Eingabe an und berechnen dann (so wie in den vorherigen Beispielen) schrittweise die partiellen Ableitungen bezüglich der Eingabe bis wir bei der Ausgabe angelangt sind. ("forward accumulation")
   $\Rightarrow$ Damit können wir die Ableitungen sämtlicher Knoten bezüglich der Eingabe effizient berechnen.
2. Wir werten zuerst den Berechnungsgraphen aus ("forward-pass") und speichern die Zwischenergebnisse. Dann fangen wir bei der Ausgabe an und berechnen schrittweise die partiellen Ableitungen der Ausgabe bezüglich der einzelnen Knoten bis wir bei der Eingabe angelangt sind ("backward accumulation" / "Backpropagation").
   $\Rightarrow$ Damit können wir die Ableitungen der Ausgabe bezüglich sämtlicher Knoten des Berechnungsgraphen effizient berechnen. Diese Ableitungen der Ausgabe (z.B. des Losses) bezüglich der Knoten (z.B. Netzwerkparameter) sind genau, was wir zum Gradientenabstieg benötigen.

Backpropagation - Beispiel 1



Im Backpropagation-Algorithmus starten wir bei der Ausgabe (typischerweise dem Loss L) und gehen dann rückwärts vor. Zunächst berechnen wir also $\frac{\partial L}{\partial g}$ und $\frac{\partial L}{\partial h}$ und dann:

$$\frac{\partial L}{\partial f} = \frac{\partial L}{\partial g}\frac{\partial g}{\partial f} + \frac{\partial L}{\partial h}\frac{\partial h}{\partial f}$$

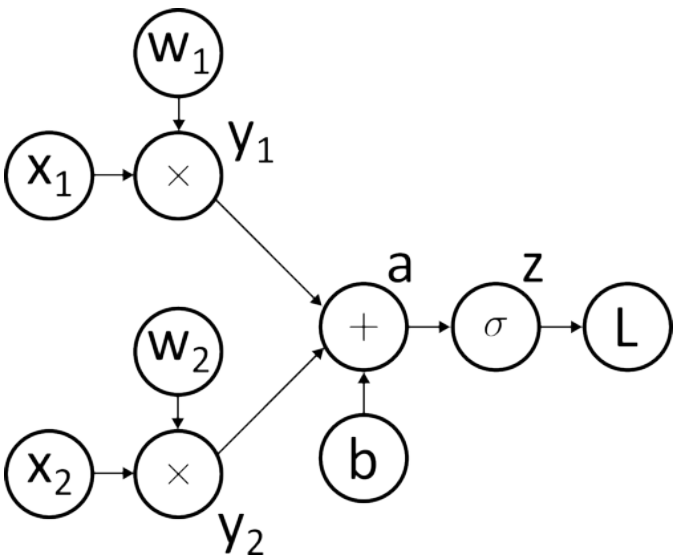
$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial x}; \frac{\partial L}{\partial y} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial y}; \frac{\partial L}{\partial z} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial z}$$

$$\frac{\partial L}{\partial s} = \frac{\partial L}{\partial x}\frac{\partial x}{\partial s} + \frac{\partial L}{\partial y}\frac{\partial y}{\partial s} + \frac{\partial L}{\partial z}\frac{\partial z}{\partial s}$$

(Wenn wir nun wieder alle Komponenten in $\frac{\partial L}{\partial s}$ einsetzen und ausmultiplizieren würden, würden wir wieder für alle 6 Pfade von s nach L einen Term erhalten.)

Backpropagation - Beispiel 2

Betrachten wir nun den Backpropagation Algorithmus für ein einzelnes Neuron:



$$\frac{\partial L}{\partial z} = \text{Ableitung des Losses}$$

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial z} \underbrace{\frac{\partial z}{\partial a}}_{\text{Ableitung der Aktivierungsfunktion}}$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial a} \underbrace{\frac{\partial a}{\partial b}}_{=1}; \frac{\partial L}{\partial y_i} = \frac{\partial L}{\partial a} \underbrace{\frac{\partial a}{\partial y_i}}_{=1}$$

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial y_i} \underbrace{\frac{\partial y_i}{\partial w_i}}_{=x_i}; \frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial y_i} \underbrace{\frac{\partial y_i}{\partial x_i}}_{=w_i}$$

Anmerkung: Damit man z.B. $\frac{\partial L}{\partial z}$, $\frac{\partial z}{\partial a}$ oder $\frac{\partial y_i}{\partial w_i}$ auswerten kann, muss man im "forward-pass" bereits die Ergebnisse für z, a oder x_i zwischenspeichern.
$\Rightarrow$ die Gradienten von L bezüglich b und w_i können wir jetzt zur Optimierung des Neurons mit Gradient-Descent verwenden.

Ableitungen in mehreren Dimensionen - Jacobi-Matrix

Im vorherigen Kapitel zu Feed-Forward Networks haben wir gesehen, wie wir ein ganzes Layer mit mehreren Neuronen elegant mit Hilfe von Vektoren und Matrizen beschreiben konnten:

$$\boldsymbol{a} = W\boldsymbol{x} + \boldsymbol{b}$$

$$\boldsymbol{z} = f(\boldsymbol{a})$$

Um Ableitungen von Abbildungen in mehreren Dimensionen effizient zu berechnen können wir Jacobi-Matrizen verwenden.

Jacobi-Matrix (Reminder)

Für eine Funktion f mit Vektoren als Inputs und Outputs (d.h. $\boldsymbol{f} : \mathbb{R}^m \to \mathbb{R}^n$) können wir eine *Jacobi-Matrix* $\boldsymbol{J_f}(x) \in \mathbb{R}^{n \times m}$ einführen, welche die *partiellen Ableitungen* von $\boldsymbol{f}$ an der Stelle x enthält:

$$(\boldsymbol{J_f}(\boldsymbol{x}))_{i,j} = \frac{\partial}{\partial x_j} \boldsymbol{f}(\boldsymbol{x})_i$$

Kettenregel in mehreren Dimensionen

In mehreren Dimensionen kann die Kettenregel mit Hilfen von Matrixmultiplikationen von Jacobi-Matrizen berechnet werden. Sei z.B. $f : \mathbb{R}^n \to \mathbb{R}^m$ und $g : \mathbb{R}^m \to \mathbb{R}^k$, dann ist $g \circ f : \mathbb{R}^n \to \mathbb{R}^k$. Damit folgt für die Ableitung der Verkettung von $g \circ f(\boldsymbol{x})$:

$$\frac{\partial}{\partial \boldsymbol{x}} g(f(\boldsymbol{x})) = \frac{\partial g}{\partial f} \frac{\partial f}{\partial \boldsymbol{x}} = J_g(f(\boldsymbol{x})) J_f(\boldsymbol{x}) = J_{g \circ f}(\boldsymbol{x})$$

Wenn wir nun die Jacobi-Matrizen $J_g(f(\boldsymbol{x})) \in \mathbb{R}^{k \times m}$ und $J_f(\boldsymbol{x}) \in \mathbb{R}^{m \times n}$ miteinander multiplizieren, so erhalten wir die Jacobi-Matrix der Verkettung $J_{g \circ f}(\boldsymbol{x}) \in \mathbb{R}^{k \times n}$.

Beispiel: Berechnung des Gradienten bezüglich x in einem Neuronalen Netz

Sei $nn(x) : \mathbb{R}^n \to \mathbb{R}^m$ ein neuronales Netz und $L : \mathbb{R}^m \to \mathbb{R}^1$ eine Loss-funktion, dann errechnet sich der Gradient bezüglich $\boldsymbol{x}$ wie folgt:

$$\frac{\partial}{\partial \boldsymbol{x}} L(nn(\boldsymbol{x})) = \frac{\partial L}{\partial nn} \frac{\partial nn}{\partial \boldsymbol{x}} = J_L(nn(\boldsymbol{x})) J_{nn}(\boldsymbol{x}) = J_{L \circ nn}(\boldsymbol{x}) \in \mathbb{R}^{1 \times n}$$

Üblicherweise besteht das Neuronale Netz aus einer Verkettung von vielen Funktionen (z.B.: Matrixmultiplikationen order Aktivierungsfunktionen). Wenn wir diese Funktionen als $l_1, l_2, ..., l_N$ schreiben und $nn(x) = (l_1 \circ l_2 \circ ... \circ l_N)(\boldsymbol{x})$ setzen, dann können wir $J_{nn}(\boldsymbol{x})$ in weitere Multiplikationen von Jacobi-Matrizen für die einzelnen l_i unterteilen:

$$J_{nn}(\boldsymbol{x}) = J_{l_1}((l_2 \circ ... \circ l_N)(\boldsymbol{x})) \cdot J_{l_2}((l_3 \circ ... \circ l_N)(\boldsymbol{x})) \cdot ... \cdot J_{l_N}(\boldsymbol{x})$$

Beispiele für Jacobi-Matrizen

Jacobi-Matrix für L_2-Loss

Betrachten wir ein Regressionsproblem, wobei der quadrierte Fehler eines Samples (x, y) zwischen einem Ausgabe-Vektor $\hat{\boldsymbol{y}}(x) \in \mathbb{R}^n$ und einem Ground-Truth Vektor $\boldsymbol{y} \in \mathbb{R}^n$ minimiert werden soll :

$$L_2(\hat{\boldsymbol{y}}(x), \boldsymbol{y}) = \sum_{i=1}^{n} (\hat{y}_i(x) - y_i)^2$$

Dieser Fehler kann als Funktion $L_2 : \mathbb{R}^n \to \mathbb{R}^1$ aufgefasst werden. Damit ergibt sich für die Jacobi-Matrix des L_2-Losses an der Stelle $\hat{\boldsymbol{y}}(x)$:

$$J_{L_2}(\hat{\boldsymbol{y}}(x)) = [2(\hat{y}_1(x) - y_1), 2(\hat{y}_2(x) - y_2), ..., 2(\hat{y}_n(x) - y_n)] \in \mathbb{R}^{1 \times n}$$

Beispiele für Jacobi-Matrizen

Jacobi-Matrix für L_1-Loss

Betrachten wir ein Regressionsproblem, wobei der absolute Fehler eines Samples (x, y) zwischen einem Ausgabe-Vektor $\hat{\boldsymbol{y}}(x) \in \mathbb{R}^n$ und einem Ground-Truth Vektor $\boldsymbol{y} \in \mathbb{R}^n$ minimiert werden soll :

$$L_1(\hat{\boldsymbol{y}}(x), y) = \sum_{i=1}^{n} |\hat{y}_i(x) - y_i|$$

Dieser Fehler kann als Funktion $L_1 : \mathbb{R}^n \to \mathbb{R}^1$ aufgefasst werden. Damit ergibt sich für die Jacobi-Matrix des L_1-Losses an der Stelle $\hat{\boldsymbol{y}}(x)$:

$$J_{L_1}(\hat{\boldsymbol{y}}(x)) = [\text{sign}\,(\hat{y}_1(x) - y_1)\,, \text{sign}\,(\hat{y}_2(x) - y_2)\,, ..., \text{sign}\,(\hat{y}_n(x) - y_n)] \in \mathbb{R}^{1 \times n}$$

wobei sign (x) wie folgt definiert ist:

$$\text{sign}\,(x) = \begin{cases} +1 & \text{wenn x } > \ 0 \\ 0 & \text{wenn x } = \ 0 \\ -1 & \text{wenn x } < \ 0 \end{cases}$$

Beispiele für Jacobi-Matrizen

Jacobi-Matrix für Kreuzentropie-Loss

Betrachten wir ein Klassifizierungsproblem mit 4 Klassen ([0,1,2,3]), wobei die Loss-Funktion durch Kreuz-Entropie vorgegeben ist:

$$L_{ce}(p(\hat{y}|x), y) = -\log(p(\hat{y} = y|x))$$

Hier ist $p(\hat{y}|x) = (p(\hat{y} = 0|x), p(\hat{y} = 1|x), p(\hat{y} = 2|x), p(\hat{y} = 3|x))$ eine bedingte Wahrscheinlichkeitsfunktion über die verschiedenen Klassen und y ein Ground-Truth Klassenlabel. Wäre z.B. $y = 2$, dann ergibt sich für die Jacobian von L an der Stelle $p(\hat{y}|x)$:

$$J_{L_{ce}}(p(\hat{y}|x)) = [0, 0, -1/p(\hat{y} = 2|x), 0] \in \mathbb{R}^{1\times 4}$$

Beispiele für Jacobi-Matrizen

Aktivierungsfunktion

Sei $f : \mathbb{R}^n \to \mathbb{R}^n$ eine beliebige elementweise angewendete Aktivierungsfunktion, dann ist die Jacobi-Matrix duch eine Diagonalmatrix gegeben, welche die jeweiligen Ableitungen entält:

$$J_f(\boldsymbol{x}) = \begin{bmatrix} f'(x_1) & & 0 \\ & \ddots & \\ 0 & & f'(x_n) \end{bmatrix}$$

Beispiele für Jacobi-Matrizen

Softmax Aktivierungsfunktion

$$\text{softmax}(\boldsymbol{x}) = \frac{\exp(x_i)}{\sum_j \exp(x_j)} = \begin{pmatrix} s_1 \\ s_2 \\ \dots \\ s_n \end{pmatrix}$$

Man kann zeigen, dass die Jacobian der Softmax-Funktion wie folgt berechnet werden kann:

$$J_{\text{softmax}}(\boldsymbol{x}) = \begin{bmatrix} s_1(1-s_1) & -s_1 s_2 & \dots & -s_1 s_n \\ -s_2 s_1 & s_2(1-s_2) & \dots & -s_2 s_n \\ & & \ddots & \\ -s_n s_1 & -s_n s_2 & \dots & s_n(1-s_n) \end{bmatrix}$$

Beispiele für Jacobi-Matrizen

Matrix-Multiplikation und Vektor-Addition

Sei eine Abbildung $\boldsymbol{a}(W, \boldsymbol{b}, \boldsymbol{x})$ gegeben durch eine Matrixmultiplikation einer Eingabe $\boldsymbol{x} \in \mathbb{R}^n$ mit einer Gewichtsmatrix $W \in \mathbb{R}^{m \times n}$ und einer Addition mit Biaswerten $\boldsymbol{b} \in \mathbb{R}^m$:

$$\boldsymbol{a}(W, \boldsymbol{b}, \boldsymbol{x}) = W\boldsymbol{x} + \boldsymbol{b}$$

dann sind die Jacobi-Matrizen für die Ableitungen nach $\boldsymbol{x}$ und $\boldsymbol{b}$ gegeben durch:

$$J_{\boldsymbol{a}}(\boldsymbol{x}) = W$$
$$J_{\boldsymbol{a}}(\boldsymbol{b}) = \mathbb{I}_m$$

Beispiele für Jacobi-Matrizen

Matrix-Multiplikation und Vektor-Addition

Sei eine Abbildung $\boldsymbol{a}(W, \boldsymbol{b}, \boldsymbol{x})$ gegeben durch eine Matrixmultiplikation einer Eingabe $\boldsymbol{x} \in \mathbb{R}^n$ mit einer Gewichtsmatrix $W \in \mathbb{R}^{m \times n}$ und einer Addition mit Biaswerten $\boldsymbol{b} \in \mathbb{R}^m$:

$$\boldsymbol{a}(W, \boldsymbol{b}, \boldsymbol{x}) = W\boldsymbol{x} + \boldsymbol{b} = \sum_{j=1}^{n} w_{ij} x_j + b_i$$

Die Ableitung von a_i nach w_{kl} kann wie folgt berechnet werden:

$$J_{\boldsymbol{a}}(W) = \frac{\partial a_i}{\partial w_{kl}} = \begin{cases} 0 & i \neq k \\ x_l & \text{sonst} \end{cases} = \delta_{ik} x_l = \mathbb{I}_m \otimes \boldsymbol{x}$$

... hier haben wir nun 3 Indizes (i, k, l). $J_{\boldsymbol{a}}(W)$ ist also ein Tensor 3. Stufe. Dies können wir auch mit dem äußeren Produkt $(\mathbb{I}_m \otimes \boldsymbol{x})$ ausdrücken. Bei einer Multiplikation dieses Tensors mit dem Gradienten des Losses L nach $\boldsymbol{a}$ $(\frac{\partial L}{\partial \boldsymbol{a}} \in \mathbb{R}^{1 \times m})$ entsteht ein $1 \times m \times n$ Tensor, welcher die partiellen Ableitungen für alle Einträge in W enthält.

Beispiel: Kettenregel für MLP

Betrachten wir nun noch einmal ein einfaches Neuronales Netz, welches eine Eingabe $\boldsymbol{x} \in \mathbb{R}^n$ auf eine Ausgabe $\hat{\boldsymbol{y}} \in \mathbb{R}^m$ wie folgt abbildet:

$$\begin{aligned}
\boldsymbol{a}^{(1)} &= W^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)} \\
\boldsymbol{z}^{(1)} &= f^{(1)}(\boldsymbol{a}^{(1)}) && \text{Layer 1} \\
\boldsymbol{a}^{(2)} &= W^{(2)}\boldsymbol{z}^{(1)} + \boldsymbol{b}^{(2)} \\
\boldsymbol{z}^{(2)} &= f^{(2)}(\boldsymbol{a}^{(2)}) && \text{Layer 2} \\
\boldsymbol{a}^{(3)} &= W^{(3)}\boldsymbol{z}^{(2)} + \boldsymbol{b}^{(3)} \\
\hat{\boldsymbol{y}} &= f^{\text{out}}(\boldsymbol{a}^{(3)}) && \text{Ausgabe} \\
L_2(\hat{\boldsymbol{y}}(x), \boldsymbol{y}) &= \sum_{i=1}^{m} (\hat{y}_i(x) - y_i)^2 && \text{Loss}
\end{aligned}$$

Im letzten Schritt verwenden wir einen L_2-Loss um $\hat{\boldsymbol{y}}$ an die Ground-Truth $\boldsymbol{y}$ anzunähern. Mit Hilfe der Kettenregel lässt sich nun der Gradient des Losses bezüglich der Eingabe $\boldsymbol{x}$ des Netzwerkes so berechnen:

$$\frac{\partial}{\partial \boldsymbol{x}} L_2 = J_{L_2}(\hat{\boldsymbol{y}}) \cdot J_{f^{\text{out}}}(\boldsymbol{a}^{(3)}) \cdot W^{(3)} \cdot J_{f^{(2)}}(\boldsymbol{a}^{(2)}) \cdot W^{(2)} \cdot J_{f^{(1)}}(\boldsymbol{a}^{(1)}) \cdot W^{(1)}$$

Gradienten bezüglich Bias-Vektoren

... zur Optimierung des Netzes sind wir jedoch an den Gradienten bezüglich der Parameter des Netzwerkes (Bias-Vektoren und Gewichts-Matrizen) interessiert! Auch hier können wir die Kettenregel verwenden.

Gradienten bezüglich Bias-Vektoren

$$\frac{\partial}{\partial \boldsymbol{b}^{(3)}} L_2 = J_{L_2}(\hat{\boldsymbol{y}}) \cdot J_{f^{\text{out}}}(\boldsymbol{a}^{(3)})$$

$$\frac{\partial}{\partial \boldsymbol{b}^{(2)}} L_2 = J_{L_2}(\hat{\boldsymbol{y}}) \cdot J_{f^{\text{out}}}(\boldsymbol{a}^{(3)}) \cdot W^{(3)} \cdot J_{f^{(2)}}(\boldsymbol{a}^{(2)})$$

$$\frac{\partial}{\partial \boldsymbol{b}^{(1)}} L_2 = J_{L_2}(\hat{\boldsymbol{y}}) \cdot J_{f^{\text{out}}}(\boldsymbol{a}^{(3)}) \cdot W^{(3)} \cdot J_{f^{(2)}}(\boldsymbol{a}^{(2)}) \cdot W^{(2)} \cdot J_{f^{(1)}}(\boldsymbol{a}^{(1)})$$

(Die Identitätsmatrizen $\mathbb{I}_m$ für $J_{\boldsymbol{a}^{(i)}}(\boldsymbol{b}^{(i)})$ können weggelassen werden)

Gradienten bezüglich Gewichts-Matrizen

Gradienten bezüglich Gewichts-Matrizen

Daraus ergibt sich für die Gradienten bezüglich der Gewichtsmatrizen:

$$\frac{\partial}{\partial W^{(3)}} L_2 = \left(J_{L_2}(\hat{\boldsymbol{y}}) \cdot J_{f^{\text{out}}}(\boldsymbol{a}^{(3)}) \right) \otimes \boldsymbol{z}^{(2)}$$

$$\frac{\partial}{\partial W^{(2)}} L_2 = \left(J_{L_2}(\hat{\boldsymbol{y}}) \cdot J_{f^{\text{out}}}(\boldsymbol{a}^{(3)}) \cdot W^{(3)} \cdot J_{f^{(2)}}(\boldsymbol{a}^{(2)}) \right) \otimes \boldsymbol{z}^{(1)}$$

$$\frac{\partial}{\partial W^{(1)}} L_2 = \left(J_{L_2}(\hat{\boldsymbol{y}}) \cdot J_{f^{\text{out}}}(\boldsymbol{a}^{(3)}) \cdot W^{(3)} \cdot J_{f^{(2)}}(\boldsymbol{a}^{(2)}) \cdot W^{(2)} \cdot J_{f^{(1)}}(\boldsymbol{a}^{(1)}) \right) \otimes \boldsymbol{x}$$

(Die Identitätsmatrizen $\mathbb{I}_m$ in $J_{\boldsymbol{a}^{(i)}}(W^{(i)})$ können weggelassen werden)

Backpropagation-Algorithmus für MLP

Insgesamt ergibt sich damit für die Berechnung der Gradienten nach $\boldsymbol{b}^{(i)}$ und $W^{(i)}$:

$$\frac{\partial}{\partial \boldsymbol{b}^{(3)}} L_2 = \left(J_{L_2}(\hat{\boldsymbol{y}}) \cdot J_{f^{\text{out}}}(\boldsymbol{a}^{(3)}) \right)$$

$$\frac{\partial}{\partial W^{(3)}} L_2 = \left(J_{L_2}(\hat{\boldsymbol{y}}) \cdot J_{f^{\text{out}}}(\boldsymbol{a}^{(3)}) \right) \otimes \boldsymbol{z}^{(2)}$$

$$\frac{\partial}{\partial \boldsymbol{b}^{(2)}} L_2 = \left(J_{L_2}(\hat{\boldsymbol{y}}) \cdot J_{f^{\text{out}}}(\boldsymbol{a}^{(3)}) \cdot W^{(3)} \cdot J_{f^{(2)}}(\boldsymbol{a}^{(2)}) \right)$$

$$\frac{\partial}{\partial W^{(2)}} L_2 = \left(J_{L_2}(\hat{\boldsymbol{y}}) \cdot J_{f^{\text{out}}}(\boldsymbol{a}^{(3)}) \cdot W^{(3)} \cdot J_{f^{(2)}}(\boldsymbol{a}^{(2)}) \right) \otimes \boldsymbol{z}^{(1)}$$

$$\frac{\partial}{\partial \boldsymbol{b}^{(1)}} L_2 = \left(J_{L_2}(\hat{\boldsymbol{y}}) \cdot J_{f^{\text{out}}}(\boldsymbol{a}^{(3)}) \cdot W^{(3)} \cdot J_{f^{(2)}}(\boldsymbol{a}^{(2)}) \cdot W^{(2)} \cdot J_{f^{(1)}}(\boldsymbol{a}^{(1)}) \right)$$

$$\frac{\partial}{\partial W^{(1)}} L_2 = \left(J_{L_2}(\hat{\boldsymbol{y}}) \cdot J_{f^{\text{out}}}(\boldsymbol{a}^{(3)}) \cdot W^{(3)} \cdot J_{f^{(2)}}(\boldsymbol{a}^{(2)}) \cdot W^{(2)} \cdot J_{f^{(1)}}(\boldsymbol{a}^{(1)}) \right) \otimes \boldsymbol{x}$$

... wie man sieht, können die Jacobi-Matrizen wiederverwendet werden, wenn wir die Berechnung der Gradienten schrittweise vom letzten Layer zum ersten Layer durchgehen. Dadurch können die Gradienten mit hoher Genauigkeit und numerischer Effizienz berechnet werden.

Backpropagation-Algorithmus in Pytorch

```python
import torch

def identity(x):
  return x

x = torch.randn(10,1,requires_grad=True) # "Ground Truth data x"
y = torch.randn(10,1,requires_grad=False) # "Ground Truth data y"
W = [torch.randn(10,10,requires_grad=True) for i in range(3)] # Weight-Matrices
b = [torch.randn(10,1,requires_grad=True) for i in range(3)] # Bias-Vectors
f = [torch.sigmoid,torch.sigmoid,identity] # Activation functions
a = [] # activations
z = [x] # outputs of individual Layers

# loop over network layers
for k in range(3):
  a.append(torch.matmul(W[k],z[-1])+b[k]) # compute activations
  z.append(f[k](a[-1])) # apply activation function

# compute square error loss between network prediction and ground truth y
Loss = torch.sum((z[-1]-y)**2)

# compute Gradients with backpropagation algorithm
Loss.backward()

# print out Gradients
print(f"x.grad = {x.grad.data}")

for k in range(3):
  print(f"Layer {k}:")
  print(f"W[{k}].grad = {W[k].grad.data}")
  print(f"b[{k}].grad = {b[k].grad.data}")
```

Figure: Backpropagation in Pytorch ist eine Zeile Code

../deeplearn