nicht-SVM-artigen Fehlerfunktionen für Least Squares (grün gestrichelt; vgl. Kap. 4.1 ab Seite 74), Logistic Regression (rot gestrichelt; vlg. Kap. 4.5 ab Seite 99), und die Anzahl der Klassifikationsfehler (schwarz gestrichelt; vlg. Kap. 4.3 ab Seite 87).

E_{L1SVM} und E_{LRSVM} sind an ihren Knickstellen (bei $t_n y_n = 1$) nicht differenzierbar. Dieser Makel kann durch Abrunden der Rampenfunktion behoben werden, z.B. durch Subtraktion einer Alpha-Funktion $\alpha_r(z) := ze^{-rz}$ mit $\alpha_r(0) = 0$, $\alpha_r'(x) = e^{-rz} - r\alpha_r(z) = (1 - rz)e^{-rz}$ und $\alpha_r'(0) = 1$. Die abgerundete Rampenfunktion [14]

$$
\begin{aligned}
&[z]_{+,r} := \text{ramp}_r(z) := (z - \alpha_r(z)) \cdot \epsilon(z) \qquad \text{mit} \\
&[z]'_{+,r} = (1 - (1 - rz)e^{-rz}) \cdot \epsilon(z) = \begin{cases} 1 - (1 - rz)e^{-rz} & , z \geq 0 \\ 0 & , z \leq 0 \end{cases}
\end{aligned} \tag{7.31}
$$

ist damit für alle $z \in \mathbb{R}$ differenzierbar und für hinreichend großes $r \gg 1$ kaum von $[z]_+$ zu unterscheiden. Die verbesserten differenzierbaren Fehlerfunktionen sind damit

$$
E_{\text{L1SVM},r}(\mathbf{w}) = \sum_{n=1}^{N} [1 - t_n y_n]_{+,r} + \lambda ||\mathbf{w}||^2 \tag{7.32}
$$

$$
E_{\text{LRSVM},r}(\mathbf{w}) = \sum_{n=1}^{N} \ln\left(\frac{1 + e^{-t_n y_n}}{1 + e^{-1}}\right) \epsilon(1 - t_n y_n) + \lambda ||\mathbf{w}||^2 \ . \tag{7.33}
$$

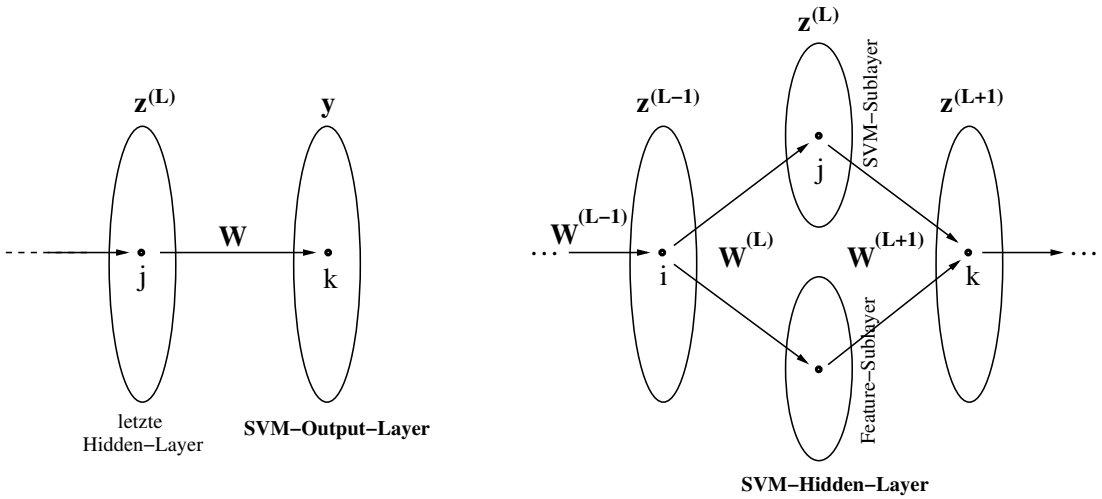
die im Grenzfall $r \to \infty$ in die ursprünglichen Fehlerfunktionen übergehen, in den Knickstellen aber die Ableitung 0 für alle $r < \infty$ behalten.

7.4.4 Backpropagation mit SVM-Layern

Falls eine SVM innerhalb eines Deep Neural Networks verwendet wird, kann man in (7.18) die Werte der Basisfunktionen $(\boldsymbol{\phi}(\mathbf{x}))_j$ mit den Aktivierungen z_j der vorigen Hidden-Layer identifizieren, sodass die SVM innerhalb der Deep Networks als einfaches lineares Neuron erscheint. Im allgemeinen Fall von K Klassen kann man eine SVM-Layer aus mehreren SVMs verwenden (siehe folgende Skizze), z.B. K SVMs, wobei die k-te SVM Klasse k gegen alle restlichen Klassen klassifiziert ("one-versus-the-rest"). [15]

[14] Aus der Definition $[z]_{+,r} := (z - \alpha_r(z)) \cdot \epsilon(z)$ nach (7.31) folgt mit der Produktregel die Ableitung $\text{ramp}_r'(z) = [z]'_{+,r} = (1 - (1 - rz)e^{-rz})\epsilon(z) + (z - \alpha_r(z))\delta(z) = (1 - (1 - rz)e^{-rz})\epsilon(z) + (0 - \alpha_r(0))\delta(z) = (1 - (1 - rz)e^{-rz})\epsilon(z)$. Man beachte, dass die Ableitung an der Knickstelle $z = 0$ eindeutig $\text{ramp}'(0) = 0$ ist.

[15] Eine SVM ist eigentlich ein 2-Klassen-Klassifikator. Gebraucht man sie "one-versus-the-rest" für K-Klassen-Probleme können Probleme auftreten, wie z.B. unbalancierte Datenverteilungen (Klasse k hat viel weniger Beispiel-Daten als die Summe der restlichen Klassen) oder unpassende Skalierung der verschiedenen Diskriminanzwerte y_k. Für eine Diskussion und mögliche Lösungen siehe Bishop (2006; S.338).



SVM-Layer als Output-Layer

Der üblichste Fall ist es, die SVMs in der Ouput-Layer zur abschließenden Klassifikation einzusetzen (siehe obige Skizze, links). Damit lassen sich die Aktivierungen der SVM durch

$$\mathbf{y} = \mathbf{W}^{\mathrm{T}}\mathbf{z}^{(L)} \qquad \text{bzw.} \qquad y_k = \sum_j W_{kj} z_j^{(L)} \tag{7.34}$$

darstellen, wobei $\mathbf{z}^{(L)}$ die Feuerraten der letzten Hidden-Layer sind und die Gewichtsmatrix $\mathbf{W} = \begin{pmatrix}\mathbf{w}_1 & \cdots & \mathbf{w}_K\end{pmatrix}$ in der k-ten Spalte den Gewichtsvektor $\mathbf{w}_k$ der k-ten SVM enthält. Da die SVM-Layer als lineare Output-Layer mit dendritischen Potentialen $a_k = y_k$ erscheint, muss im Backpropagation-Algorithmus nach Satz 5.1 auf Seite 124 die Initialisierung der Fehler-Terme $\delta_k := \frac{\partial E_n}{\partial a_k} = \frac{\partial E_n}{\partial y_k}$ in Schritt IIa einfach für die verschiedenen SVM-Fehlerfunktionen angepasst werden:

- **Hinge/L1-SVM:** Nach (7.26) ist $E_n = [1 - t_{nk} y_{nk}]_+ + \frac{\lambda}{N}||\mathbf{w}||^2$. Mit der Ableitung $[z]'_+ = \mathrm{ramp}'(z) = \epsilon(z)$ der Rampenfunktion [16] folgt mit Hilfe der Kettenregel

$$\delta_k = \frac{\partial E_n}{\partial y_k} = -t_{nk}\epsilon(1 - t_{nk} y_{nk}) = \begin{cases} -t_{nk} & , t_{nk} y_{nk} < 1 \\ 0 & , \text{sonst} \end{cases} \tag{7.35}$$

[16] Wegen $[z]_+ = z$ für $z \geq 0$ folgt $[z]'_+ = 1$ für $z > 0$. Wegen $[z]_+ = 0$ für $z < 0$ folgt $[z]'_+ = 0$ für $z < 0$. Für $z = 0$ hat $[z]_+$ einen "Knick" und ist leider nicht sinnvoll differenzierbar. Dasselbe Ergebnis erhält man auch aus $\mathrm{ramp}(z) = z\epsilon(z)$ mit $\epsilon'(z) = \delta(z)$, wobei $\delta(z)$ hier der Dirac-Stoß sei, aus der Produktregel und der Siebeigenschaft von $\delta(z)$: $\mathrm{ramp}'(z) = 1 \cdot \epsilon(z) + z \cdot \delta(z) = \epsilon(z) + 0 \cdot \delta(z) = \epsilon(z)$. Für weitere Eigenschaften der Rampenfunktion siehe auch Skript Signale und Systeme ????.

- **L2-SVM:** Nach (7.27) ist $E_n = ([1 - t_{nk}y_{nk}]_+)^2 + \frac{\lambda}{N}||\mathbf{w}||^2$ und es folgt ähnlich nach zweifachem Anwenden der Kettenregel

$$\begin{aligned}\delta_k = \frac{\partial E_n}{\partial y_k} &= 2[1 - t_{nk}y_{nk}]_+\epsilon(1 - t_{nk}y_{nk})(-t_{nk}) \\ &= -2t_{nk}[1 - t_{nk}y_{nk}]_+ = \begin{cases} 2(y_{nk} - t_{nk}) & , t_{nk}y_{nk} < 1 \\ 0 & , \text{sonst} \end{cases}\end{aligned} \tag{7.36}$$

- **LR-SVM:** Nach (7.30) ist $E_n = \left[\ln\left(\frac{1+e^{-t_n y_n}}{1+e^{-1}}\right)\right]_+ + \frac{\lambda}{N}||\mathbf{w}||^2$ und es folgt [17]

$$\begin{aligned}\delta_k = \frac{\partial E_n}{\partial y_k} &= -(1 + e^{-1})t_{nk} \cdot \sigma(t_{nk}y_{nk}) \cdot \epsilon(1 - t_{nk}y_{nk}) \\ &= \begin{cases} -(1 + e^{-1})t_{nk} \cdot \sigma(t_{nk}y_{nk}) & , t_{nk}y_{nk} < 1 \\ 0 & , \text{sonst} \end{cases}\end{aligned} \tag{7.37}$$

- Die **differenzierbaren Fehlerfunktionen** $E_{\text{L1SVM},r}$ nach (7.32) und $E_{\text{LRSVM},r}$ nach (7.33) für L1-SVM und LR-SVM könnte man ähnlich ableiten. Wir benutzen sie hier aber nur um zu bemerken, dass diese für $r \to \infty$ einerseits äquivalent zu den nicht differnzierbaren SVMs werden, und andererseits an den Knickstellen stets die Ableitung 0 haben, was konsistent mit obigen Initialisierungen von L1-SVM und LR-SVM ist.

Bemerkung: Man kann mit obigen Fehlerfunktionen natürlich auch die Gewichte $\mathbf{w}_k$ der SVMs ausschlieës lich durch Gradientenabstieg lernen. Es ist jedoch meist effizienter für die SVMs in jedem Lernschritt einen optimierte Lernalgorithmen zu verwenden (z.B. Quadratic Optimization oder Sequential Minimal Optimization (SMO), siehe Kap.???) [18], und obige Initialisierungen nur zum Berechnen der Fehlerterme δ_j in den Hidden Layern zu verwenden. Da optimierte Lernalgorithmen nur für L1-SVM und L2-SVM existieren, kann man für die restlichen SVM-Typen auch einen hybriden Ansatz wählen: In jedem Lernschritt wir zunächst etwa über SMO eine Näherung für $\mathbf{w}_k$ bestimmt, und anschließend über Gradientenabstieg oder Newtonverfahren weiter optimiert.

SVM-Layer als Hidden-Layer

Alternativ kann man die SVMs auch in einer Hidden-Layer einsetzten (siehe obige Skizze, rechts). Dies ist z.B. sinnvoll für kompositionelle Objekt-Darstellungen bei denen das

[17] Mit $\ln'(z) = 1/z$ folgt nach mehrmaligem Anwenden der Kettenregel $\frac{\partial}{\partial y_n}\left[\ln\left(\frac{1+e^{-t_n y_n}}{1+e^{-1}}\right)\right]_+ = \epsilon\left(\ln\left(\frac{1+e^{-t_n y_n}}{1+e^{-1}}\right)\right) \cdot \frac{1+e^{-1}}{1+e^{-t_{nk}y_{nk}}} e^{-t_{nk}y_{nk}}(-t_{nk}) = -(1 + e^{-1})t_{nk} \cdot \sigma(t_{nk}y_{nk}) \cdot \epsilon(1 - t_{nk}y_{nk})$.

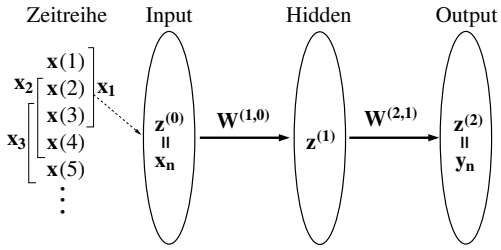
[18] Dazu verwendet man als Inputvektoren statt der $\boldsymbol{\phi}_n$ die Aktivierungsvektoren $\mathbf{z}_n^L$ der letzen Hidden-Layers, welche den Input zur SVM bildet.

Klassifikationsmodell Informationen über die Bestandteile bestimmter Objektklassen enthält, z.B. dass ein Hund aus Kopf, Rumpf, Beinen und Schwanz besteht. In diesem Fall könnte man z.B. eine oder mehrere Hidden-SVM-Layers, welche die Bestandteile erkennen sollen, parallel zu einer gewöhnlichen Hidden-Feature-Layer schalten. Falls auch die Bildbestandteile gelabelt sind kann man in jedem Lernschritt die SVMs wie oben beschrieben optimieren und andererseits deren Aktivierungen (ggf. nach einer Softmax-Operation) als Input für nachfolgende Layers verwenden, um das Gesamtobjekt zu erkennen. Für dieses Vorgehen kann man natürlich statt SVM-Layers beliebige Klassifikatoren benutzen, sofern sie geeignete Fehlerfunktionen besitzen die mit dem Backprop-Verfahren kompatibel sind.

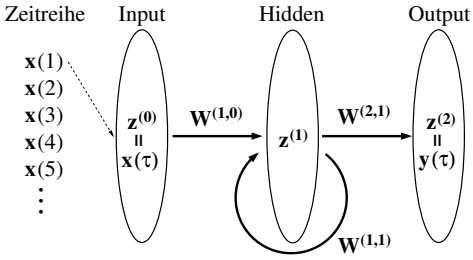
7.5 Rekurrente Netze und Backpropagation-Through-Time

Bisher haben wir unsere Betrachtung auf zyklenfreie Feed-Forward-Netzwerke beschränkt (siehe Fußnote 4 auf Seite 121). Dies war notwendig um einen klar definierten Informationsfluss sowohl für die Neuronaktivitäten (von Input- vorwärts zur Output-Schicht) als auch für die Backpropagation der Fehlersignale zu haben (von Output- zurück zur Input-Schicht). Man nimmt dabei an, dass jeder Input-Vektor $\mathbf{x}_n$ unabhängig von anderen Inputs verarbeitet werden kann, und es keine (etwa zeitlichen) Abhängigkeiten zwischen verschiedenen Inputs gibt. In vielen Anwendungen ist diese Annahme jedoch nicht erfüllt: Z.B. hängt bei der Verarbeitung von Zeitreihen-Verläufen (etwa Aktienkursen) der aktuelle Input $\mathbf{x}(\tau)$ zur Zeit τ meist stark von früheren Inputs $\mathbf{x}(\tau), \ldots, \mathbf{x}(\tau - T)$ ab. Dasselbe gilt für die zu prädizierenden Zielwerte (z.B. steigender oder fallender Kurs). Um solche Abhängigkeiten zu erfassen müss te man durch sogenanntes **Windowing** mehrere Daten-Inputs zum Netzwerk-Input $\mathbf{x}_n := \big(\mathbf{x}(\tau) \cdots \mathbf{x}(\tau - T)\big)$ zusammenfassen (siehe folgende Skizze I). Der Vorteil ist, dass wir dadurch die bisherigen Lernmethoden für Feed-Forward-Netzwerke auch auf solche Probleme anwenden können. Wesentlicher Nachteil ist, dass dadurch die Input-Dimension $(T + 1) \cdot D$ vervielfacht würde, was das Lernproblem erheblich schwieriger macht ("Curse of Dimensionality"). Ein alternativer Ansatz sind sogenannte **rekurrente Neuronale Netze (RNN)** bei denen der Output einer Hidden-Layer auf den Input rückgekoppelt wird (Skizze II). Hier hofft man, dass durch einen geeigneten Lernvorgang die rekurrenten Gewichte sich so einstellen, dass die wesentlichen Abhängigkeiten der Daten-Inputs in den Aktivitäten der Hidden Units gespeichert und für die Prädiktion nutzbar bleiben. Man spricht deshalb auch von Short-Term-Memory bzw. deutsch Kurzzeit-Gedächtnis. Da durch die rekurrenten Verbindungen das Netzwerk nicht mehr zyklenfrei ist muss der bisher besprochene Backpropagation-Algorithmus wie folgt modifiziert werden.

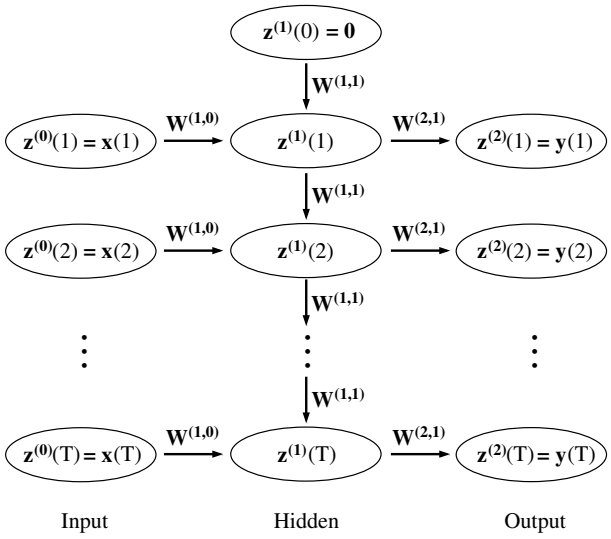
I) MLP mit Windowing



II) Rekurrentes Neuronales Netz (RNN)



III) Zeitlich entfaltetes RNN für Backpropagation through time (BPTT)



Betrachten Sie das Netzwerk in obiger Skizze II: Trotz rekurrenter Verbindungen kann man im Forward-Pass die Neuron-Aktivitäten offenbar als sequentielle Verarbeitung über die Zeit $\tau = 1, ..., T$ darstellen (vgl. die Notation von S.120 bzw. S.131): Nach Initialisieren der Hidden-Layer $\mathbf{z}^{(1)}(0) := \mathbf{0}$ gilt für Inputs $\mathbf{z}^{(0)}(\tau) := \mathbf{x}(\tau)$ in der Hidden-Layer die Rekursion

$$\begin{aligned} \mathbf{a}^{(1)}(\tau) &= \mathbf{W}^{(\mathbf{1,0})}\mathbf{z}^{(0)}(\tau-1) + \mathbf{W}^{(\mathbf{1,1})}\mathbf{z}^{(1)}(\tau-1) \qquad \text{mit} \\ \mathbf{z}^{(1)}(\tau) &= h^{(1)}(\mathbf{a}^{(1)}(\tau)) \end{aligned} \tag{7.38}$$

und für die Output-Layer $\mathbf{a}^{(2)}(\tau) = \mathbf{W}^{(\mathbf{2,1})}\mathbf{z}^{(1)}(\tau)$ und $\mathbf{z}^{(2)}(\tau) = h^{(2)}(\mathbf{a}^{(2)}(\tau)) =: \mathbf{y}(\tau)$ in jedem Zeitschritt $\tau = 1, ..., T$. Entfalten der Rekursion (7.38) ergibt die sequentiellen Verarbeitungsschritte

$$\begin{aligned} \mathbf{a}^{(1)}(1) &= \mathbf{W}^{(\mathbf{1,0})}\mathbf{z}^{(0)}(0) + \mathbf{W}^{(\mathbf{1,1})}\mathbf{z}^{(1)}(0) && \text{und} \quad \mathbf{z}^{(1)}(1) = h^{(1)}(\mathbf{a}^{(1)}(1)) \\ \mathbf{a}^{(1)}(2) &= \mathbf{W}^{(\mathbf{1,0})}\mathbf{z}^{(0)}(1) + \mathbf{W}^{(\mathbf{1,1})}\mathbf{z}^{(1)}(1) && \text{und} \quad \mathbf{z}^{(1)}(2) = h^{(1)}(\mathbf{a}^{(1)}(2)) \\ &\vdots && \vdots \\ \mathbf{a}^{(1)}(T) &= \mathbf{W}^{(\mathbf{1,0})}\mathbf{z}^{(0)}(T-1) + \mathbf{W}^{(\mathbf{1,1})}\mathbf{z}^{(1)}(T-1) && \text{und} \quad \mathbf{z}^{(1)}(T) = h^{(1)}(\mathbf{a}^{(1)}(T)) \ . \end{aligned}$$

Offenbar ist die Verarbeitung in dem rekurrenten Netzwerk also äquivalent zu der Verarbeitung in dem in Skizze III dargestellten Feed-Forward-Netzwerk mit Weight-Sharing der Gewichts-Matrizen $\mathbf{W}^{(\mathbf{1,0})}, \mathbf{W}^{(\mathbf{1,1})}, \mathbf{W}^{(\mathbf{2,1})}$ ähnlich wie in Kapitel 7.3 ab Seite 156. Wir können also im entfalteten Netzwerk von Skizze III durch gewöhnliches Backpropagation zunächst die Fehlersignale $\boldsymbol{\delta}^{(L)}(\tau)$ ausrechnen und damit nach Satz 7.3 auf

Seite 159 alle Gewichtsänderungen von "shared weights" addieren. D.h. nach Initialisieren der Fehler in den Output-Layern, z.B. $\boldsymbol{\delta}^{(2)}(\tau) := \mathbf{t}(\tau) - \mathbf{y}(\tau)$, ergibt Backpropagation über die Zeitschritte (**Backpropagation-through-time**) nach Satz 5.1.II.b auf Seite 124 in der Matrix-/Vektorschreibweise von Seite 132 für die Fehlersignale in der Hidden-Layer

$$\boldsymbol{\delta}^{(1)}(T) := {h^{(1)}}'(\mathbf{a}^{(1)}(T)) \odot \left(\mathbf{W}^{(\mathbf{2},\mathbf{1})^{\mathrm{T}}} \boldsymbol{\delta}^{(2)}(T)\right) \qquad \text{und}$$
$$\boldsymbol{\delta}^{(1)}(\tau) := {h^{(1)}}'(\mathbf{a}^{(1)}(\tau)) \odot \left(\mathbf{W}^{(\mathbf{2},\mathbf{1})^{\mathrm{T}}} \boldsymbol{\delta}^{(2)}(\tau) + \mathbf{W}^{(\mathbf{1},\mathbf{1})^{\mathrm{T}}} \boldsymbol{\delta}^{(1)}(\tau+1)\right)$$

für $\tau = T-1, T-2, \ldots, 1$. Nach Satz 7.3 auf Seite 159 ergeben sich damit für die drei Gewichtsmatrizen des RNN die partiellen Ableitungen

$$\frac{\partial E_{D,n}}{\partial W_{ji}^{(1,1)}} = \sum_{\tau=1}^{T} \delta_j^{(1)}(\tau) z_i^{(1)}(\tau-1) = \sum_{\tau=1}^{T} \boldsymbol{\delta}^{(1)}(\tau) \cdot \mathbf{z}^{(1)}(\tau-1)^{\mathrm{T}}$$
$$\frac{\partial E_{D,n}}{\partial W_{ji}^{(2,1)}} = \sum_{\tau=1}^{T} \delta_j^{(2)}(\tau) z_i^{(1)}(\tau) = \sum_{\tau=1}^{T} \boldsymbol{\delta}^{(2)}(\tau) \cdot \mathbf{z}^{(1)}(\tau)^{\mathrm{T}}$$
$$\frac{\partial E_{D,n}}{\partial W_{ji}^{(1,0)}} = \sum_{\tau=1}^{T} \delta_j^{(1)}(\tau) z_i^{(0)}(\tau) = \sum_{\tau=1}^{T} \boldsymbol{\delta}^{(1)}(\tau) \cdot \mathbf{z}^{(0)}(\tau)^{\mathrm{T}}$$

und damit Gewichtsgradient und Gewichtsupdate wie in Satz 5.1 auf Seite 124. Falls manche Zielwerte $\mathbf{t}(\tau)$ nicht zur Verfügung stehen kann man die entsprechenden Output-Zweige im entfalteten Netzwerk einfach weg lassen. Will man z.B. nur am Ende einer Sequenz eine Vorhersage machen, kann man alle Output-Layers bis auf die letzte Layer $\mathbf{z}^{(2)}(T) = \mathbf{y}(T)$ weglassen. Regularisierung etwa mit Weight-Decay kann man wie üblich durch Addition des Gewichtsgradienten zum Datengradienten realisieren (siehe (5.19) auf Seite 125). Das folgende Python-Skript implementiert den **BPTT Algorithmus** (vgl. S. 241):

```
import numpy as np

def softmax(a):  # compute softmax function for potential vector a; for numerical stability
    e_a = np.exp(a - np.max(a))      # subtract maximum potential such that max. exponent is 1
    return e_a / e_a.sum()           # return softmax function value

def forwardPropagateActivity(x_list,W10,W11,W21,b=1.0): # prop. activity x through RNN with bias b in z1
    Tau=len(x_list)                  # length of input sequence x_list (list of np.arrays)
    z_1_list,z_2_list=[],[]          # Initialize empty lists of layer activities
    z_1 = np.zeros(M,dtype='float')  # initialize hidden state with zeros
    for tau in range(Tau):           # loop over time steps tau=1,2,...,Tau (here actually 0,...,Tau-1)
        a_1 = np.dot(W10,x_list[tau])+np.dot(W11,np.append(z_1,[b])) # dendritic pot. hidden lyr
        z_1 = np.tanh(np.array(a_1,'float')) # compute new activity z_1 of hidden layer 1
        a_2 = np.dot(W21,np.append(z_1,[b])) # compute dendritic potentials of output layer a_2
        z_2 = softmax(a_2)                   # compute softmax activations for output layer
```

```
        z_1_list.append(z_1)                # append z_1 to result list
        z_2_list.append(z_2)                # append z_2 to result list
    return z_1_list,z_2_list         # return list of activities in (unfolded) layers z_1 and z_2

def backPropagateErrors(z_1_list,z_2_list,t_list,W10,W11,W21): # backpropagate error signals delta_L
    Tau,M,K=len(z_1_list),W10.shape[0],W21.shape[0] # Sequence length, hidden- and output-layer size
    delta_1_list,delta_2_list=[],[]        # initialize lists for error signals of each time step
    delta_1=np.zeros(M,dtype='float')      # initialize error signals for hidden layer
    for tau in range(Tau-1,-1,-1):         # loop backwards over layers tau=Tau,Tau-1,...,1 (minus 1)
        delta_2=np.zeros(K,dtype='float')  # initialize output errors with zeros as default values
        if not t_list[tau] is None: delta_2=z_2_list[tau]-t_list[tau]; # if available, init output errors
        alpha12=np.dot(W21.T,delta_2)[:-1] # error potentials by backprop layer 2 --> layer 1
        alpha11=np.dot(W11.T,delta_1)[:-1] # error potentials by backprop layer 1 --> layer 1
        h_prime=(1.0-np.multiply(z_1_list[tau],z_1_list[tau])) # factor (1-z_1.*z_1) is h'(a) for tanh
        delta_1=np.multiply(h_prime,alpha12+alpha11) # compute error signals in hidden layer 1
        delta_1_list.append(delta_1)       # append error signals of layer 1 to result list
        delta_2_list.append(delta_2)       # append error signals of layer 2 to result list
    delta_1_list.reverse()                 # reverse results list (for ordering tau=0,1,...,Tau-1
    delta_2_list.reverse()                 # ditto for layer 2 error signals
    return delta_1_list,delta_2_list       # return lists of error signals for each (unfolded) layer

def doLearningStep(W10,W11,W21,xn_list,tn_list,eta,lmbda_by_N=0,b=1.0): # do one BPTT step...
    z_1_list,z_2_list=forwardPropagateActivity(xn_list,W10,W11,W21,b); # forward prop. of input sequence
    delta_1 ,delta_2 =backPropagateErrors(z_1_list,z_2_list,tn_list,W10,W11,W21); # get errors by bckprop
    nablaED_10 = np.zeros(shape=W10.shape) # initialize gradients for weights W10
    nablaED_11 = np.zeros(shape=W11.shape) # initialize gradients for weights W11
    nablaED_21 = np.zeros(shape=W21.shape) # initialize gradients for weights W21
    for tau in range(len(xn_list)): # loop over shared weights in the enfolded network
        nablaED_10 = nablaED_10+np.outer(delta_1[tau],xn_list[tau]) # compute gradients...
        if tau>0: nablaED_11 = nablaED_11+np.outer(delta_1[tau],np.append(z_1_list[tau-1],[b])) # step1?
        else:     nablaED_11 = nablaED_11+np.outer(delta_1[tau],np.append(np.zeros(W11.shape[0]),[b]))
        nablaED_21 = nablaED_21+np.outer(delta_2[tau],np.append(z_1_list[tau],[b]))
    W10=W10*(1.0-lmbda_by_N*eta)-eta*nablaED_10     # weight update with "weight decay" regularization
    W11=W11*(1.0-lmbda_by_N*eta)-eta*nablaED_11     # weight update with "weight decay" regularization
    W21=W21*(1.0-lmbda_by_N*eta)-eta*nablaED_21     # update update with "weight decay" regularization
    return W10,W11,W21, nablaED_10,nablaED_11,nablaED_21 # return new weights (and gradients to verify)

def getError(W10,W11,W21,X,T,lmbda=0,b=1.0): # crossentropy error function over data (X,T) for RNN
    N,K=len(X),W21.shape[0]                 # Number of sequences,output layer size
    E=0.0;                                  # initialize error with 0
    for n in range(N):                      # test all data vectors
        y_list=forwardPropagateActivity(X[n],W10,W11,W21,b)[1]; # get output y for input X[n]
        t_list=T[n]                         # get actual target vector sequence ("one hot" coded)
        e=[-t_list[tau][i]*np.log(y_list[tau][i])\
           for tau in range(len(y_list)) for i in range(K)\
           if not t_list[tau] is None and t_list[tau][i]>0 and y_list[tau][i]>0] # errors for each component
        E=E+np.sum(e)                               # add sum of component errors to total error
    if lmbda>0:                                     # weight decay regularization?
        EW10=np.sum(np.sum(np.multiply(W10,W10))) # sum of squared weights for W10
        EW11=np.sum(np.sum(np.multiply(W11,W11))) # sum of squared weights for W11
        EW21=np.sum(np.sum(np.multiply(W21,W21))) # sum of squared weights for W21
```

```
        E=E+0.5*lmbda*(EW10+EW11+EW21)            # add weight error
    return E;                                  # return final error

# ******************************************************
# Main program
# ******************************************************
if __name__ == '__main__':
    # (i) Create training data: Generate time series being a sum of two sinus functions
    # The learning goal is to predict if values will increase by at least th percents after ph time steps
    N,Tau  =10,100     # number of sequences, length of each sequence
    D,K    =1,3        # dimensions of input vectors, dimensions of outputs (one-hot!)
    X,T=[],[]          # initialize data and targets with empty lists
    f1, f2 =0.05, 0.1  # frequencies of the two sinus functions (in cycles per time step)
    ph =5              # prediction horizon (predict after ph steps)
    th =0.05           # threshold for classifying "increasing" or "decreasing" (e.g., 0.05=5%)
    sig_offset,sig_phase,sig_noise=0.05,5,0.1 # standard deviations for offset, phase, value of the sinus
    #sig_offset,sig_phase,sig_noise=0.1,5,0.05 # standard deviations for offset, phase, value of the sinus
    t_inc = np.array([1,0,0])  # one-hot codeword for predicting "increasing" value >= v*(1+th)
    t_eq  = np.array([0,1,0])  # one-hot code for predicting "equal" value between v*(1-th) and v*(1+th)
    t_dec = np.array([0,0,1])  # one-hot codeword for predicting "decreasing" value <= v*(1-th)
    for n in range(N):
        offset = np.abs(np.random.normal(0,sig_offset,1)[0]) # offset noise --> added to sinus function
        phase  = np.random.normal(0,sig_phase,1)[0]          # phase noise  --> phase shift for sinus
        tau    = np.array(range(Tau),'float')                # time samples tau=0,...,Tau-1
        x      = np.sin(2*np.pi*f1*tau+phase)+np.sin(2*np.pi*f2*tau+phase)+2.5  # time sequence values
        x      = x+offset+np.abs(np.random.normal(0,sig_noise,Tau)) # add offset and individual noise
        x_list = [np.array([x[tau],1.0]) for tau in range(Tau)] # x as list of np-arrays with input bias
        t_list = [t_eq for tau in range(Tau)] # init sequence of target prediction values with defaults
        for tau in range(0,Tau-ph):
            if x_list[tau+ph][0]>=(x_list[tau][0]*(1.0+th)):t_list[tau]=t_inc   # target "increasing"
            if x_list[tau+ph][0]<=(x_list[tau][0]*(1.0-th)):t_list[tau]=t_dec   # target "decreasing"
        X.append(x_list)       # append new data sequence to data list
        T.append(t_list)       # append new target sequence to target list

    # (ii) Generate Recurrent neural network (RNN) and train it by Backpropagation Through Time (BPTT)
    M=3           # number of hidden units (not counting bias unit)
    b=1.0         # include bias unit for hidden layer?
    eta0=0.0005    # initial learning rate
    etadecay=500.0 # learning decay: learning rate is eta = eta0/(1+epoch/etadecay) --> 0
    lmbda=0       # regularization coefficient
    nEpochs=5000   # number of training epochs
    W10=1.0*(np.random.rand(M,D+1)-0.5) # initialize weights W10 randomly
    W11=1.0*(np.random.rand(M,M+1)-0.5) # initialize weights W11 randomly
    W21=1.0*(np.random.rand(K,M+1)-0.5) # initialize weights W21 randomly
    E=getError(W10,W11,W21,X,T,0,b)     # compute initial data error for the random network...
    print("initial data error E=",E)    # and print it
    for epoch in range(nEpochs):   # loop over learning epochs
        datc,errc = 0,0            # initialize classification error counters with zero
        for n in range(N):         # loop over all training data
            xn=X[n]                # n-th data sequence
            tn=T[n]                # n-th target sequence
```

```
            yn=forwardPropagateActivity(xn,W10,W11,W21,b)[1] # get prediction for sequence xn
            for tau in range(len(xn)-ph):                    # loop over all time steps of yn
                t,y=np.argmax(tn[tau]),np.argmax(yn[tau])    # target and predicition
                if t!=y: errc=errc+1                         # prediction error? then increase counter
                datc=datc+1                                  # count also total number of predictions
            W10,W11,W21=doLearningStep(W10,W11,W21,xn,tn,eta0/(1.0+epoch/etadecay),lmbda/N,b)[:3] # BPTT
        E=getError(W10,W11,W21,X,T,0,b) # compute data error
        print("after epoch",epoch,"training data error E=",E,"prediction error=", errc,"/",datc)
        #if epoch%10==0: checkDataGradients(W10,W11,W21,X[0],T[0],delta=1e-8,eps=1e-8,b=b)
```

In dem Hauptprogramm werden Zeitreihen von fester Länge T erzeugt (Summe von zwei Sinus-Funktionen unterschiedlicher Frequenz, mit zufälligem Offset, Phase und additivem Noise). Das Ziel ist eine kontinuierliche Vorhersage ob `ph`=5 Schritte in der Zukunft der Wert um mindestens `th`=5 Prozent ansteigt, abfällt oder im Intervall $\pm 5\%$ gleich bleibt. Nach 5000 Lern-Epochen wir ein Prediction Error von 10.6 Prozent auf den Trainingsdaten erreicht.

Bemerkung: Bei der Implementierung BPTT und noch komplizierterer Verfahren stellt sich immer die Frage nach der Korrektheit des Verfahrens. Insbesondere sollte man immer überprüfen ob der durch das Backpropagation-Verfahren berechnete Gradient mit dem tatsächlichen Gradienten der Fehlerfunktion übereinstimmen. Den tatsächlichen Gradienten kann man über die Definition der partiellen Ableitungen als Differenzenquotienten

$$\frac{\partial E(\cdots, w_{ji}, \cdots)}{\partial w_{ji}} = \lim_{\epsilon \to 0} \frac{E(\cdots, w_{ji} + \epsilon, \cdots) - E(\cdots, w_{ji} - \epsilon, \cdots)}{2\epsilon} \tag{7.39}$$

bestimmen. Nach Entfernung des Kommentarzeichens in der letzten Zeile obigen Skripts kann der (Teil-)Gradient z.B. mit folgender Python-Funktion überprüft werden:

```
def checkDataGradients(W10,W11,W21,xn,tn,delta=1e-6,eps=1e-6,b=1.0): # verify partial gradients
    # (i) compute gradient for W10
    m,n=W10.shape
    nablaW10=np.zeros(shape=(m,n))
    W10_copy=np.array(W10)
    for i in range(m):
        for j in range(n):
            W10_copy[i,j]=W10[i,j]+delta
            nablaW10[i,j]=getError(W10_copy,W11,W21,[xn],[tn],0,b)
            W10_copy[i,j]=W10[i,j]-delta
            nablaW10[i,j]=(nablaW10[i,j]-getError(W10_copy,W11,W21,[xn],[tn],0,b))/(2*delta)
            W10_copy[i,j]=W10[i,j]
    # (ii) compute gradient for W11
    m,n=W11.shape
    nablaW11=np.zeros(shape=(m,n))
    W11_copy=np.array(W11)
    for i in range(m):
        for j in range(n):
            W11_copy[i,j]=W11[i,j]+delta
```

```
        nablaW11[i,j]=getError(W10,W11_copy,W21,[xn],[tn],0,b)
        W11_copy[i,j]=W11[i,j]-delta
        nablaW11[i,j]=(nablaW11[i,j]-getError(W10,W11_copy,W21,[xn],[tn],0,b))/(2*delta)
        W11_copy[i,j]=W11[i,j]
# (iii) compute gradient for W21
m,n=W21.shape
nablaW21=np.zeros(shape=(m,n))
W21_copy=np.array(W21)
for i in range(m):
    for j in range(n):
        W21_copy[i,j]=W21[i,j]+delta
        nablaW21[i,j]=getError(W10,W11,W21_copy,[xn],[tn],0,b)
        W21_copy[i,j]=W21[i,j]-delta
        nablaW21[i,j]=(nablaW21[i,j]-getError(W10,W11,W21_copy,[xn],[tn],0,b))/(2*delta)
        W21_copy[i,j]=W21[i,j]
# (iv) get gradients via backpropagation
nablaED_10,nablaED_11,nablaED_21=doLearningStep(W10,W11,W21,xn,tn,0,0,b)[3:]
# (v) evaluate difference
len10=np.linalg.norm(nablaW10.flat)
len11=np.linalg.norm(nablaW11.flat)
len21=np.linalg.norm(nablaW21.flat)
diff10=np.linalg.norm((nablaED_10-nablaW10).flat)
diff11=np.linalg.norm((nablaED_11-nablaW11).flat)
diff21=np.linalg.norm((nablaED_21-nablaW21).flat)
relerr10=diff10/len10
relerr11=diff11/len11
relerr21=diff21/len21
if np.abs(relerr10)>eps or  np.abs(relerr11)>eps or np.abs(relerr21)>eps:
        print("Gradient difference: len10=",len10,"diff10=",diff10,"relerr10=",relerr10)
        print("Gradient difference: len11=",len11,"diff11=",diff11,"relerr11=",relerr11)
        print("Gradient difference: len21=",len21,"diff21=",diff21,"relerr21=",relerr21)
```

Weitere Bemerkungen zu BPTT:

I) Der Rechenaufwand obiger BPTT-Implementierung pro Lernepoche ist $O(NTS)$ wobei N die Anzahl der Trainings-Sequenzen, T die Länge einer Sequenz und S die Anzahl der Synapsen im Netzwerk ist.

II) Für große Sequenzlänge T ergibt sich das Problem der **Vanishing Gradients** (deutsch: verschwindende Gradienten): Da in jedem Backpropagation-Schritt ein Faktor $h'(a) < 1$ multipliziert wird (z.B. $\sigma' < 0.3$ nach Figur auf S. 238; und $\tanh'(a) < 0.5$ für $|a| > 1$) tendieren die Fehlersignale und damit die partiellen Ableitungen in den vorderen Layern exponentiell mit T gegen 0, zumindest falls die Fehlerpotentiale nicht zu schnell mit $O(1)$ wachsen. Dadurch werden die Gewichte in den vorderen Layern praktisch nicht mehr verändert und das Lernen hört auf.

III) Ähnlich kann bei großen Fehlerpotentialen $h'(a) \cdot \alpha > 1$ das Problem der **Exploding Gradients** (deutsch: explodierende Gradienten) auftreten: Da in jedem Backpropagation-Schritt das Fehlerpopential multipliziert wird, tendieren

die Fehlersignale und damit die partiellen Ableitungen exponentiell gegen ∞. Dadurch bräuchte man für sinnvollen Gradientenabstieg sehr kleine Schrittweiten, wodurch aber das Lernen in den hinteren Layern zum Stillstand käme.

IV) Diese Probleme treten allgemein bei **Deep Learning** in **Deep Neural Networks** auf, wenn der Backpropagation-Algorithmus auf ein "tiefes" Netzwerk aus sehr vielen Layern angewandt wird (wie es z.B. bei BPTT durch die zeitliche Entfaltung entsteht). Man kann versuchen sie dadurch in den Griff zu bekommen, dass die Gewichtsmatrizen mit geeigneten Zufallsgewichten initialisiert werden, deren Stärke so gewählt wird, dass das Produkt aus h' und den Fehlerpotentialen im Mittel 1 ergibt (mit möglichst kleiner Varianz), und es somit beim Rückpropagieren zu keinem (oder nur einem geringen) Verstärkungs- oder Abschwächungs-Effekt kommt.

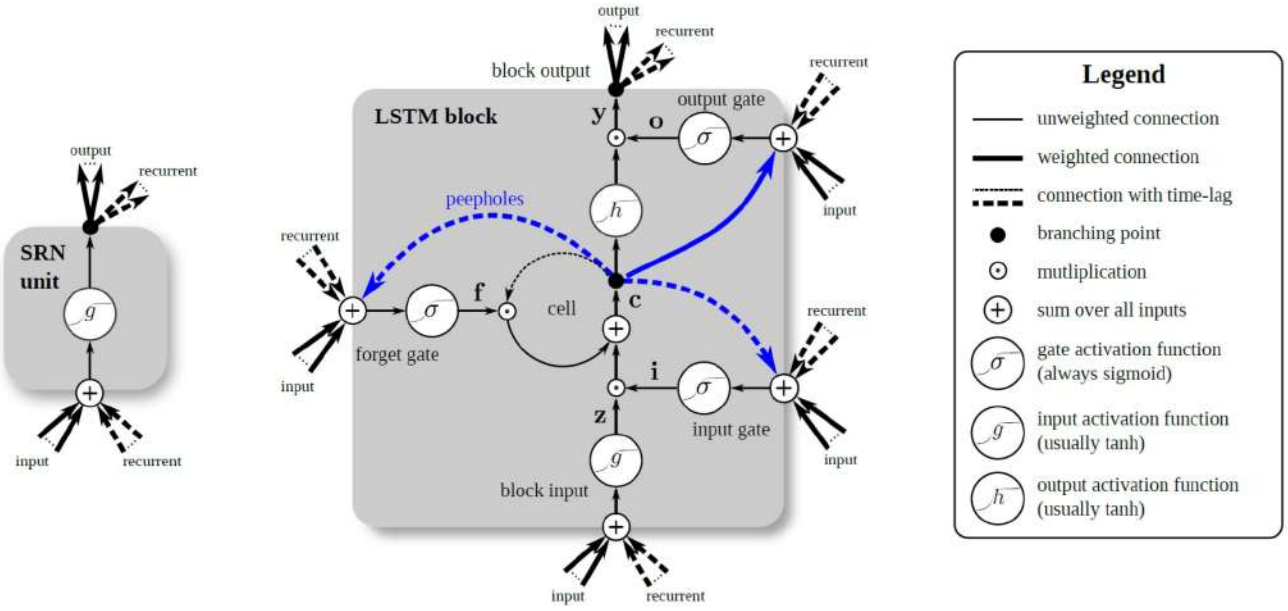
V) Bei BPTT kann man diese Probleme auch dadurch in den Griff bekommen, dass man lange Sequenzen in viele kürzere Sequenzen zerschneidet: Will man z.B. eine Prognose nur aufgrund der letzten 10 Schritte einer Zeitreihe machen, kann man eine lange Sequenz der Länge $L_s > 10$ in $L_s - 10$ Sequenzen der Länge $T = 10$ mit jeweils einem Zielwert t zerlegen, und damit das entfaltete Netzwerk auf Tiefe $T = 10$ reduzieren. Allerdings können damit auch nur noch Abhängigkeiten bis T Schritte in die Vergangenheit zur Prognose genutzt werden.

VI) **Truncated Backpropagation Through Time (TBPTT)** mit Parametern k_1 und k_2 ist eine beliebte Variante: Dabei werden ähnlich wie vorher Sequenzen der Länge L_s in Sequenzen der Länge $T := k_1$ zerschnitten. Anders als bei BPTT wird aber Backpropagation im entfalteten Netzwerk bereits nach $k_2 \leq k_1$ Layers $T, T-1, \ldots, T-k_2+1$ abgebrochen (engl. *truncated*). Da in der Hidden-Layer $z^{(1)}$ Information über die ganze Sequenz gespeichert bleibt, kann man damit prinzipiell noch Abhängigkeiten bis k_1 Schritte in die Vergangenheit lernen, obwohl das entfaltete Netzwerk nur Tiefe k_2 hat. Der Nachteil ist allerdings, dass mit diesem Verfahren der tatsächliche Gradient nur noch approximiert wird. Somit ist nicht mehr garantiert dass eine kleine Gewichtsänderung in Gradientenrichtung zu einer Verkleinerung des Fehlers führt. Außerdem kann man eine fehlerhafte Implementierung nicht mehr mit obiger Python-Funktion erkennen!

VII) Bei Abhängigkeiten die sehr weit in die Vergangenheit reichen versagen TBPTT und BPTT in der Regel: In diesen Fällen sind alternative Netzwork-Modelle wie etwa **Long-Short-Term-Memory (LSTM)** wesentlich erfolgversprechender (siehe Kapitel 7.6). Hierbei wird versucht die zu entfaltende Hidden-Layer durch lineare Units mit $h(a) = a$ und $h'(a) = 1$ mit Rückkopplungs-Gewicht 1 als sogenanntes **Constant Error Carrousel** zu realisieren, um damit die Verstärkung bzw. Abschwächung der Fehlersignale und damit das Problem der exploding/vanishing Gradients zu eliminieren.

7.6 Long-Short-Term-Memory (LSTM)

Long-Short-Term-Memory (LSTM) wurde entwickelt um die im vorigen Abschnitt beschriebenen Problemen von einfachen RNN mit BPTT zu lösen. Es definiert wie in der folgenden Skizze dargestellt eine Hidden Unit eines rekurrentes Netzes nicht durch ein einzelnes simples Neuron wie im RNN (linke Skizze, SRN unit = simple recurrent network unit), sondern durch zusätzliche kompliziertere Elemente wie diverse Gate-Funktionen mit multiplikative Verbindungen (siehe Kap. 7.2), ein Constant Error Carrousel (CEC) und sogenannte Peephole-Verbindungen (rechte Skizze, LSTM block).



Tatsächlich existieren mittlerweile verschiedene Varianten von LSTM mit unterschiedlichen Gates und mit oder ohne Peepholes. Wir behandeln im folgenden die beliebteste Variante, das sogenannte **Vanilla LSTM**, das für die meisten Anwendungsfälle sehr gute Ergebnisse liefert, wobei wir die Notation wie in obiger Skizze verwenden. [19] Die oben dargestellte LSTM-Layer kann beliebig in ein neuronales Netzwerk eingebunden sein. Hierbei bekommt die LSTM-Layer Input von einer vorgeschalteten Input-Layer $\mathbf{x}$ der Größe D, und gibt seinen Output über die LSTM-Output-Layer $\mathbf{y}$ der Größe M an nachgeschaltete Schichten des Netzwerkes weiter. Wir behandeln zunächst wie im

[19] Klaus Greff; Rupesh Kumar Srivastava; Jan Koutník; Bas R. Steunebrink; Jürgen Schmidhuber (2015): LSTM: A Search Space Odyssey. IEEE Transactions on Neural Networks and Learning Systems 28(10):2222–2232.

Forward-Pass die Aktivität von der Input-Layer $\mathbf{x}^\tau$ zur Zeit τ die entsprechende Aktivität der Output-Layer $\mathbf{y}^\tau$ definiert. [20] Dafür nehmen wir an, dass die LSTM-Layer aus M LSTM-Blocks besteht (zur Erinnerung: Jeder LSTM-Block entspricht einer Hidden Unit der LSTM-Layer). Die Verbindungen zur LSTM-Layer und innerhalb der LSTM-Layer lassen sich durch folgende **Gewichtsmatrizen und -vektoren** beschreiben: [21]

I) Input-Gewichte: $\mathbf{W_z}, \mathbf{W_i}, \mathbf{W_f}, \mathbf{W_o} \in \mathbb{R}^{M \times D}$

II) Rekurrente Gewichte: $\mathbf{R_z}, \mathbf{R_i}, \mathbf{R_f}, \mathbf{R_o} \in \mathbb{R}^{M \times M}$

III) Peephole-Gewichte: $\mathbf{p_i}, \mathbf{p_f}, \mathbf{p_o} \in \mathbb{R}^{M}$

IV) Bias-Gewichte: $\mathbf{b_z}, \mathbf{b_i}, \mathbf{b_f}, \mathbf{b_o} \in \mathbb{R}^{M}$

Damit kann man die Aktivitäts-Ausbreitung im **Forward-Pass** wie folgt schreiben:

$$
\begin{array}{lll}
\text{Block-Input :} & \mathbf{z}^\tau := g(\bar{\mathbf{z}}^\tau) \quad \text{für} & \bar{\mathbf{z}}^\tau := \mathbf{W_z}\mathbf{x}^\tau + \mathbf{R_z}\mathbf{y}^{\tau-1} + \mathbf{b_z} \\
\text{Input-Gate :} & \mathbf{i}^\tau := \sigma(\bar{\mathbf{i}}^\tau) \quad \text{für} & \bar{\mathbf{i}}^\tau := \mathbf{W_i}\mathbf{x}^\tau + \mathbf{R_i}\mathbf{y}^{\tau-1} + \mathbf{p_i} \odot \mathbf{c}^{\tau-1} + \mathbf{b_i} \\
\text{Forget-Gate :} & \mathbf{f}^\tau := \sigma(\bar{\mathbf{f}}^\tau) \quad \text{für} & \bar{\mathbf{f}}^\tau := \mathbf{W_f}\mathbf{x}^\tau + \mathbf{R_f}\mathbf{y}^{\tau-1} + \mathbf{p_f} \odot \mathbf{c}^{\tau-1} + \mathbf{b_f} \\
\text{Cell-State :} & \mathbf{c}^\tau := \mathbf{z}^\tau \odot \mathbf{i}^\tau + \mathbf{c}^{\tau-1} \odot \mathbf{f}^\tau & \\
\text{Output-Gate :} & \mathbf{o}^\tau := \sigma(\bar{\mathbf{o}}^\tau) \quad \text{für} & \bar{\mathbf{o}}^\tau := \mathbf{W_o}\mathbf{x}^\tau + \mathbf{R_o}\mathbf{y}^{\tau-1} + \mathbf{p_o} \odot \mathbf{c}^{\tau} + \mathbf{b_o} \\
\text{Block-Output:} & \mathbf{y}^\tau := h(\mathbf{c}^\tau) \odot \mathbf{o}^\tau & \qquad (7.40)
\end{array}
$$

Hierbei ist die Gate-Sigmoide $\sigma(a) := \frac{1}{1+e^{-a}} \in (0;1)$ die logistische Sigmoid-Funktion von Lemma **??** auf Seite **??**. Die Block-Input/Ouput Sigmoiden sind üblicherweise $g(a) := h(a) := \tanh(a) \in (-1;1)$. Der Kern eines LSTM-Blocks besteht aus dem **Cell-State** $\mathbf{c}^\tau$, der als lineares Neuron mit einer einzelnen rekurrenten Verbindung mit Gewicht 1 auf sich selbst das auf Seite 174 erwähnte **Constant Error Carrousel (CEC)** zur Verhinderung der Vanishing/Exploding Gradients realisiert. Ansonsten bekommt jeder Cell-State nur noch einen einzelnen durch das Input-Gate modulierten Input vom Block-Input. Aufgabe des Cell-States ist es Information über frühere Inputs beliebig lange zu speichern. Dabei kann der Cell-State nur durch den Block-Input überschrieben werden, wenn das Input-Gate öffnet. Ähnlich moduliert die Multiplikation mit dem Forget-Gate die Stärke der rekurrenten Verbindung auf Werte ≤ 1 und bestimmt damit ob und wie schnell die gespeicherte Information wieder vergessen werden kann. Das Output-Gate bestimmt schließlich ob und wieviel der gespeicherten Information zu

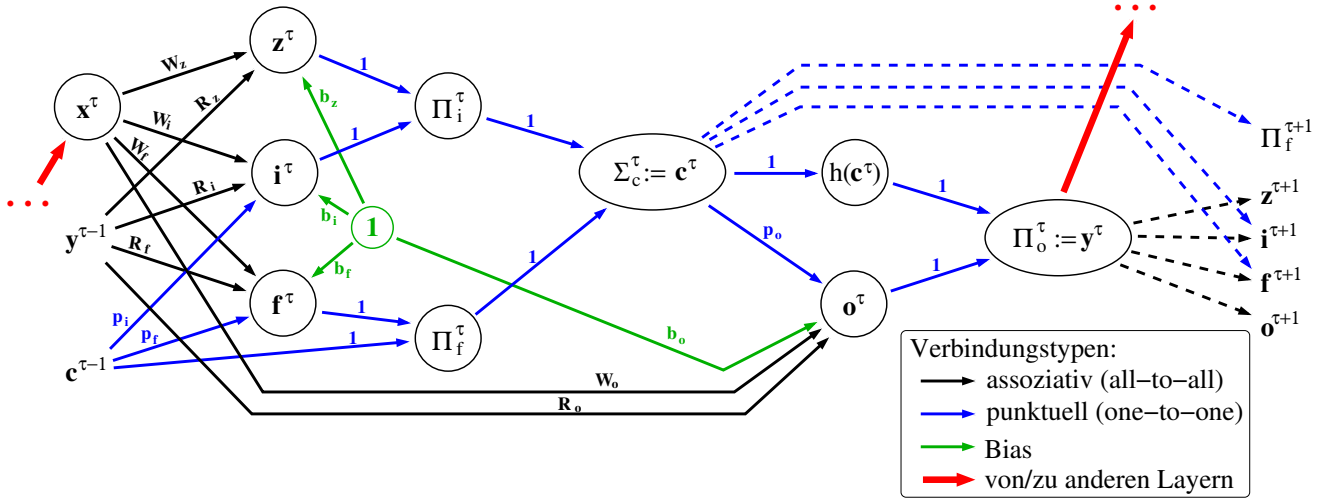
[20] Um übermäßig viele Klammern zu vermeiden schreiben wir anders als in Kap. 7.5 nun $\mathbf{x}^\tau$ statt $\mathbf{x}(\tau)$.

[21] Anders als bisher in den Feed-Forward-Netzwerken integrieren wir den Bias nicht in die Gewichtsmatrizen, sondern stellen ihn als Eigenschaft der jeweiligen Neuron-Layer als Vektor dar. Dadurch vermeidet man, dass ein Neuron mehrere Bias-Inputs bekommt. Andererseits muss man deutlich mehr Gewichts-Variablen definieren.

einer bestimmten Zeit zur Output-Layer fließen kann. Die Peephole Connections liefern zusätzliche Information über den Speicher-Zustand an die drei Gates, damit diese ihr Öffnen bzw. Schließen mit dem Speicherzustand besser koordinieren können.

Backward-Pass: Im Backward-Pass werden wieder mit Backpropagation Through time die Gewichte im zeitlich entfalteten Netzwerk angepasst (vgl. Kap. 7.5 ab Seite 167). Die folgende Skizze zeigt einen etwas aufsührlicheren Ausschnitt aus dem entfalteten LSTM-Netzwerk mit allen involvierten Verbindungen und mit expliziter Darstellung des Cell States $\sigma_c^\tau := \mathbf{c}^\tau$ als additiver Unit, welche über 1:1-Verbindungen Inputs von zwei zusätzlichen multiplikativen Units $\pi_i^\tau := \mathbf{z}^\tau \odot \mathbf{i}^\tau$ und $\pi_f^\tau := \mathbf{c}^{\tau-1} \odot \mathbf{f}^\tau$ erhält. Außerdem stellen wir auch den Block-Output explizit als multiplikative Unit $\pi_o^\tau := \mathbf{y}^\tau$ dar.

Zeitlich entfaltete LSTM–Layer (zur Zeit τ) für BPTT



Durch diese Darstellung können wir auf einfache Weise die in den vorigen Kapiteln erarbeiteten Regeln zur Rückpropagierung der Fehlersignale ins Spiel bringen:

I) Für "normale" synaptische Verbindungen zwischen Layern mit einfachen sigmoiden Aktivierungsfunktionen gilt die übliche Standard-Rekursion nach Satz 5.1.IIb auf Seite 124.

II) Hat ein Neuron bzw. eine Neuron-Layer Vorwärts-Verbindungen zu verschiedenen Layern, so addieren sich deren rückpropagierten Fehlersignale an diesem Neuron bzw. in dieser Layer (siehe (7.2) auf Seite 152 bzw. (5.10) auf Seite 123; siehe auch (5.38) auf Seite 132 für eine Darstellung in Matrix-Schreibweise).

III) Multipliziert ein Pi-Neuron zwei Faktoren, so ist das rückpropagierte Fehlersignal des Pi-Neurons zu einem Faktor gleich dem Produkt aus Fehlersignal des Pi-

Neurons multipliziert mit der Sigmoid-Ableitung des einen Faktors multipliziert mit dem anderen Faktor (siehe (7.15) auf Seite 156).

IV) Weight Sharing: Wird ein Gewichts-Parameter für mehrere (physikalische) Synapsen verwendet, so addieren sich deren partiellen Ableitungen bzw. Gewichtsänderungen (siehe Satz 7.3 auf Seite 159).

Sei nun $\boldsymbol{\delta}^\tau_{\text{inj}}$ der sogenannte "Injected Error" definiert als die Summe aller Fehlersignal von der LSTM-Layer nachgeschalteten Netzwerk-Schichten zur Zeit τ, d.h. $\boldsymbol{\delta}^\tau_{\text{inj}}$ ist die Summe aller Fehlersignale die über den dicken roten Pfeil an $\mathbf{y}^\tau$) rückpropagiert werden. Konsequentes Anwenden obiger Backpropagation-Regeln ergibt damit die folgenden Fehlersignale in dem entfalteten LSTM-Netzwerk (von rechts nach links):

$$
\begin{aligned}
\boldsymbol{\delta}^\tau_{\mathbf{y}} &= \boldsymbol{\delta}^\tau_{\text{inj}} + \mathbf{R_z}^{\mathrm{T}}\boldsymbol{\delta}^{\tau+1}_{\mathbf{z}} + \mathbf{R_i}^{\mathrm{T}}\boldsymbol{\delta}^{\tau+1}_{\mathbf{i}} + \mathbf{R_f}^{\mathrm{T}}\boldsymbol{\delta}^{\tau+1}_{\mathbf{f}} + \mathbf{R_o}^{\mathrm{T}}\boldsymbol{\delta}^{\tau+1}_{\mathbf{o}} \quad \text{(Regel I,II)} \\
\boldsymbol{\delta}^\tau_{\mathbf{o}} &= \boldsymbol{\delta}^\tau_{\mathbf{y}} \odot \sigma'(\bar{\mathbf{o}}^\tau) \odot h(\mathbf{c}^\tau) \quad \text{(Regel III)} \\
\boldsymbol{\delta}^\tau_{h(\mathbf{c})} &= \boldsymbol{\delta}^\tau_{\mathbf{y}} \odot h'(\mathbf{c}^\tau) \odot \mathbf{o}^\tau \quad \text{(Regel III)} \\
\boldsymbol{\delta}^\tau_{\mathbf{c}} &= \boldsymbol{\delta}^\tau_{h(\mathbf{c})} + \mathbf{p_o} \odot \boldsymbol{\delta}^\tau_{\mathbf{o}} + \boldsymbol{\delta}^{\tau+1}_{\boldsymbol{\Pi}_f} \odot 1 \odot \mathbf{f}^{\tau+1} \quad \text{(Regel I,II,III)} \\
&\quad + \mathbf{p_i} \odot \boldsymbol{\delta}^{\tau+1}_{\mathbf{i}} + \mathbf{p_f} \odot \boldsymbol{\delta}^{\tau+1}_{\mathbf{f}} \quad \text{(Regel I,II)} \\
\boldsymbol{\delta}^\tau_{\boldsymbol{\Pi}_\mathbf{i}} &= \boldsymbol{\delta}^\tau_{\mathbf{c}} \quad \text{(Regel I)} \\
\boldsymbol{\delta}^\tau_{\boldsymbol{\Pi}_\mathbf{f}} &= \boldsymbol{\delta}^\tau_{\mathbf{c}} \quad \text{(Regel I)} \\
\boldsymbol{\delta}^\tau_{\mathbf{z}} &= \boldsymbol{\delta}^\tau_{\boldsymbol{\Pi}_\mathbf{i}} \odot g'(\bar{\mathbf{z}}^\tau) \odot \mathbf{i}^\tau \quad \text{(Regel III)} \\
\boldsymbol{\delta}^\tau_{\mathbf{i}} &= \boldsymbol{\delta}^\tau_{\boldsymbol{\Pi}_\mathbf{i}} \odot \sigma'(\bar{\mathbf{i}}^\tau) \odot \mathbf{z}^\tau \quad \text{(Regel III)} \\
\boldsymbol{\delta}^\tau_{\mathbf{f}} &= \boldsymbol{\delta}^\tau_{\boldsymbol{\Pi}_\mathbf{f}} \odot \sigma'(\bar{\mathbf{f}}^\tau) \odot \mathbf{c}^{\tau-1} \quad \text{(Regel III)} \\
\boldsymbol{\delta}^\tau_{\mathbf{x}} &= \mathbf{W_z}^{\mathrm{T}}\boldsymbol{\delta}^\tau_{\mathbf{z}} + \mathbf{W_i}^{\mathrm{T}}\boldsymbol{\delta}^\tau_{\mathbf{i}} + \mathbf{W_f}^{\mathrm{T}}\boldsymbol{\delta}^\tau_{\mathbf{f}} + \mathbf{W_o}^{\mathrm{T}}\boldsymbol{\delta}^\tau_{\mathbf{o}} \quad \text{(Regel I,II)}
\end{aligned}
$$

Einsetzen von $\boldsymbol{\delta}^\tau_{h(\mathbf{c})}$, $\boldsymbol{\delta}^\tau_{\boldsymbol{\Pi}_\mathbf{i}}$, $\boldsymbol{\delta}^\tau_{\boldsymbol{\Pi}_\mathbf{f}}$ in die anderen Gleichungen ergibt etwas kompakter

$$
\begin{aligned}
\boldsymbol{\delta}^\tau_{\mathbf{y}} &= \boldsymbol{\delta}^\tau_{\text{inj}} + \mathbf{R_z}^{\mathrm{T}}\boldsymbol{\delta}^{\tau+1}_{\mathbf{z}} + \mathbf{R_i}^{\mathrm{T}}\boldsymbol{\delta}^{\tau+1}_{\mathbf{i}} + \mathbf{R_f}^{\mathrm{T}}\boldsymbol{\delta}^{\tau+1}_{\mathbf{f}} + \mathbf{R_o}^{\mathrm{T}}\boldsymbol{\delta}^{\tau+1}_{\mathbf{o}} \\
\boldsymbol{\delta}^\tau_{\mathbf{o}} &= \boldsymbol{\delta}^\tau_{\mathbf{y}} \odot \sigma'(\bar{\mathbf{o}}^\tau) \odot h(\mathbf{c}^\tau) \\
\boldsymbol{\delta}^\tau_{\mathbf{c}} &= \boldsymbol{\delta}^\tau_{\mathbf{y}} \odot h'(\mathbf{c}^\tau) \odot \mathbf{o}^\tau + \mathbf{p_o} \odot \boldsymbol{\delta}^\tau_{\mathbf{o}} + \boldsymbol{\delta}^{\tau+1}_{\mathbf{c}} \odot \mathbf{f}^{\tau+1} + \mathbf{p_i} \odot \boldsymbol{\delta}^{\tau+1}_{\mathbf{i}} + \mathbf{p_f} \odot \boldsymbol{\delta}^{\tau+1}_{\mathbf{f}} \\
\boldsymbol{\delta}^\tau_{\mathbf{z}} &= \boldsymbol{\delta}^\tau_{\mathbf{c}} \odot g'(\bar{\mathbf{z}}^\tau) \odot \mathbf{i}^\tau \\
\boldsymbol{\delta}^\tau_{\mathbf{i}} &= \boldsymbol{\delta}^\tau_{\mathbf{c}} \odot \sigma'(\bar{\mathbf{i}}^\tau) \odot \mathbf{z}^\tau \\
\boldsymbol{\delta}^\tau_{\mathbf{f}} &= \boldsymbol{\delta}^\tau_{\mathbf{c}} \odot \sigma'(\bar{\mathbf{f}}^\tau) \odot \mathbf{c}^{\tau-1} \\
\boldsymbol{\delta}^\tau_{\mathbf{x}} &= \mathbf{W_z}^{\mathrm{T}}\boldsymbol{\delta}^\tau_{\mathbf{z}} + \mathbf{W_i}^{\mathrm{T}}\boldsymbol{\delta}^\tau_{\mathbf{i}} + \mathbf{W_f}^{\mathrm{T}}\boldsymbol{\delta}^\tau_{\mathbf{f}} + \mathbf{W_o}^{\mathrm{T}}\boldsymbol{\delta}^\tau_{\mathbf{o}}
\end{aligned} \tag{7.41}
$$

wobei man das letze Fehlersignal $\boldsymbol{\delta}^\tau_{\mathbf{x}}$ nur benötigt wenn es vor der Input-Layer $\mathbf{x}$ mindestens eine weitere Layer gibt, deren Gewichte zu $\mathbf{x}$ trainiert werden müssen. Ansonsten kann man durch Anwenden von Satz 5.1.III auf Seite 124 in Verbindung mit Regel IV

die Gradienten für alle Gewichte in Matrixnotation als Summen von äußeren Produkten bestimmen: [22]

$$\begin{aligned}
\nabla_{\mathbf{W_z}} E_D &= \sum_{\tau=1}^{T} \boldsymbol{\delta}_{\mathbf{z}}^{\tau} \cdot \mathbf{x}^{\tau\mathrm{T}} & \nabla_{\mathbf{R_z}} E_D &= \sum_{\tau=2}^{T} \boldsymbol{\delta}_{\mathbf{z}}^{\tau} \cdot \mathbf{y}^{\tau-1\mathrm{T}} & \nabla_{\mathbf{b_z}} E_D &= \sum_{\tau=1}^{T} \boldsymbol{\delta}_{\mathbf{z}}^{\tau} \\
\nabla_{\mathbf{W_i}} E_D &= \sum_{\tau=1}^{T} \boldsymbol{\delta}_{\mathbf{i}}^{\tau} \cdot \mathbf{x}^{\tau\mathrm{T}} & \nabla_{\mathbf{R_i}} E_D &= \sum_{\tau=2}^{T} \boldsymbol{\delta}_{\mathbf{i}}^{\tau} \cdot \mathbf{y}^{\tau-1\mathrm{T}} & \nabla_{\mathbf{b_i}} E_D &= \sum_{\tau=1}^{T} \boldsymbol{\delta}_{\mathbf{i}}^{\tau} \\
\nabla_{\mathbf{W_f}} E_D &= \sum_{\tau=1}^{T} \boldsymbol{\delta}_{\mathbf{f}}^{\tau} \cdot \mathbf{x}^{\tau\mathrm{T}} & \nabla_{\mathbf{R_f}} E_D &= \sum_{\tau=2}^{T} \boldsymbol{\delta}_{\mathbf{f}}^{\tau} \cdot \mathbf{y}^{\tau-1\mathrm{T}} & \nabla_{\mathbf{b_f}} E_D &= \sum_{\tau=1}^{T} \boldsymbol{\delta}_{\mathbf{f}}^{\tau} \\
\nabla_{\mathbf{W_o}} E_D &= \sum_{\tau=1}^{T} \boldsymbol{\delta}_{\mathbf{o}}^{\tau} \cdot \mathbf{x}^{\tau\mathrm{T}} & \nabla_{\mathbf{R_o}} E_D &= \sum_{\tau=2}^{T} \boldsymbol{\delta}_{\mathbf{o}}^{\tau} \cdot \mathbf{y}^{\tau-1\mathrm{T}} & \nabla_{\mathbf{b_o}} E_D &= \sum_{\tau=1}^{T} \boldsymbol{\delta}_{\mathbf{o}}^{\tau} \\
\nabla_{\mathbf{p_i}} E_D &= \sum_{\tau=2}^{T} \boldsymbol{\delta}_{\mathbf{i}}^{\tau} \odot \mathbf{c}^{\tau-1} & \nabla_{\mathbf{p_f}} E_D &= \sum_{\tau=2}^{T} \boldsymbol{\delta}_{\mathbf{f}}^{\tau} \odot \mathbf{c}^{\tau-1} & \nabla_{\mathbf{p_o}} E_D &= \sum_{\tau=1}^{T} \boldsymbol{\delta}_{\mathbf{o}}^{\tau} \odot \mathbf{c}^{\tau}
\end{aligned} \tag{7.42}$$

wobei T die Sequenzlänge ist. Beachten Sie dass hier die "Gradienten" wie etwa $\nabla_{\mathbf{W_z}} E_D$ keine Zeilenvektoren sind, sondern Matrizen mit derselben Größe wie die jeweilige Gewichtematrix und in jeder Komponente die entsprechende partielle Ableitung steht, d.h. beispielsweise $(\nabla_{\mathbf{W_z}} E_D)_{ji} := \frac{\partial E_D}{\partial (\mathbf{W_z})_{ji}}$ oder $(\nabla_{\mathbf{b_z}} E_D)_j := \frac{\partial E_D}{\partial (\mathbf{b_z})_j}$. D.h. beim Gewichtsupdate mit Gradientenabstieg kann man einfach ein negatives Vielfaches obiger Teil-"Gradienten" zur Gewichtsmatrix addieren, z.B.

$$\Delta \mathbf{W_z} := -\eta \nabla_{\mathbf{W_z}} E_D, \qquad \Delta \mathbf{b_z} := -\eta \nabla_{\mathbf{b_z}} E_D, \qquad \ldots \tag{7.43}$$

wobei $\Delta \mathbf{W_z}, \Delta \mathbf{b_z}, \ldots$ die Gewichtsänderungen für die Gewichte $\mathbf{W_z}, \mathbf{b_z}, \ldots$ und η die Lernrate ist. Regularisierung etwa mit Weight-Decay kann man wie üblich durch Addition des Gewichtsgradienten zum Datengradienten realisieren (siehe (5.19) auf Seite 125). Das folgende Python-Skript implementiert den **LSTM Algorithmus**:

```
import numpy as np

def softmax(a):        # compute softmax function for potential vector a; for numerical stability
    e_a = np.exp(a - np.max(a)) # subtract maximum potential such that max. exponent is 1
    return e_a / e_a.sum()      # return softmax function value

def sigma(a):          # compute logistic sigmoid function
    return 1.0/(1.0+np.exp(-a)) # return function value
```

[22] Man beachte, dass man $\delta_j z_i = (\boldsymbol{\delta} \cdot \mathbf{z}^{\mathrm{T}})_{ji}$ als Komponente eines äußeren Produkts schreiben kann. Das äußere Produkt entspricht der Matrizenmultiplikation eines Spaltenvektors mit einem Zeilenvektor und ergibt eine Matrix. D.h. die resultierende Matrix enthält alle partiellen Ableitungen für den Gradienten.

```
def sigma_prime(a,z=None):    # compute first derivative of logistic sigmoid function
    if z is None: z=sigma(a)
    return np.multiply(z,1.0-z) # return function value

def g(a):              # tanh sigmoid function
    return np.tanh(a)          # return function value

def g_prime(a,z=None): # compute first derivative of tanh
    if z is None: z=np.tanh(a)  # compute activity
    return 1.0-np.multiply(z,z) # return function value

def h(a):              # tanh sigmoid function
    return np.tanh(a)          # return function value

def h_prime(a,z=None): # compute first derivative of tanh
    if z is None: z=np.tanh(a)  # compute activity
    return 1.0-np.multiply(z,z) # return function value

def forwardPropagateActivity(x_list,W): # prop. activity x through LSTM with weights W
    Tau,M=len(x_list),W['Wz'].shape[0]  # length of inp. sequence x_list (list of np.arrays) and #units
    Z_list=[{} for tau in range(Tau)]# initialize list of empty dicts of layer activities
    c,y = np.zeros(M,dtype='double'),np.zeros(M,dtype='double')  # initialize  cell state and outputs
    for tau in range(Tau):           # loop over time steps tau=1,2,...,Tau (here actually 0,...,Tau-1)
        x = x_list[tau]                                                                  # Sequence Input
        z = g(np.dot(W['Wz'],x)+np.dot(W['Rz'],y)+W['bz'])                               # Block Input
        i = sigma(np.dot(W['Wi'],x)+np.dot(W['Ri'],y)+np.multiply(W['pi'],c)+W['bi']) # Input Gate
        f = sigma(np.dot(W['Wf'],x)+np.dot(W['Rf'],y)+np.multiply(W['pf'],c)+W['bf']) # Forget Gate
        c = np.multiply(z,i)+np.multiply(c,f)                                            # Cell State
        o = sigma(np.dot(W['Wo'],x)+np.dot(W['Ro'],y)+np.multiply(W['po'],c)+W['bo']) # Output-Gate
        y = np.multiply(h(c),o)                                                          # Block Output
        z2 = softmax(np.dot(W['W21'],y)+W['bz2'])  # additional output layer (above LSTM)
        Z=Z_list[tau]                # get reference to dict in activity result list
        Z['x'],Z['z'],Z['i'],Z['f' ]=x,z,i,f       # store activities of step tau in dict
        Z['c'],Z['o'],Z['y'],Z['z2']=c,o,y,z2      # ditto
    return Z_list                    # return list of activities in (unfolded) LSTM layers (and z2)

def backPropagateErrors(Z_list, t_list, W): # backpropagate error signals
    Tau,M,K=len(Z_list),W['Wz'].shape[0],W['W21'].shape[0] # Seq. length, LSTM- and output-layer size
    delta_list=[{} for tau in range(Tau)]   # init. list of empty dicts of error signals for each step
    npz=np.zeros(M,dtype='double')          # zero array for initializing LSTM errors signals
    dc,dz,di,df,do=npz,npz,npz,npz,npz      # initialize error signals of LSTM layer (as needed)
    for tau in range(Tau-1,-1,-1):          # loop backwards over layers tau=Tau,Tau-1,...,1 (minus 1)
        # (i) first get some neuron activities
        Z=Z_list[tau]                       # activity dict for step tau
        o,c,z,i,f,z2 = Z['o'],Z['c'],Z['z'],Z['i'],Z['f'],Z['z2'] # activities at time tau
        f_tp1,c_tm1 = npz,npz               # initialize activities at other times with zeros
        if tau<(Tau-1): f_tp1 = Z_list[tau+1]['f']  # activity f at time tau+1
        if tau>0      : c_tm1 = Z_list[tau-1]['c']  # activity c at time tau-1
        # (ii) do update steps for error signals delta according to equations in lecture manuscript...
        dz2=np.zeros(K,dtype='double')      # initialize output errors with zeros as default values
        if not t_list[tau] is None: dz2=z2-t_list[tau]; # if available, init output errors
```

```
        dinj=np.dot(W['W21'].T,dz2)         # injected error is just backprop from output layer z2
        dy=dinj+np.dot(W['Rz'].T,dz)+np.dot(W['Ri'].T,di)+np.dot(W['Rf'].T,df)+np.dot(W['Ro'].T,do)
        do=np.multiply(np.multiply(dy,sigma_prime(None,o)),h(c))
        dc=np.multiply(np.multiply(dy,h_prime(c)),o)+np.multiply(W['po'],do)+np.multiply(dc,f_tp1)\
            +np.multiply(W['pi'],di)+np.multiply(W['pf'],df)
        dz=np.multiply(np.multiply(dc,g_prime(None,z)),i)
        di=np.multiply(np.multiply(dc,sigma_prime(None,i)),z)
        df=np.multiply(np.multiply(dc,sigma_prime(None,f)),c_tm1)
        dx=np.dot(W['Wz'].T,dz)+np.dot(W['Wi'].T,di)+np.dot(W['Wf'].T,df)+np.dot(W['Wo'].T,do)
        delta=delta_list[tau]               # get reference to dict in error result list
        delta['y'],delta['o'],delta['c'],delta['z'] =dy,do,dc,dz  # store errors of step tau in dict
        delta['i'],delta['f'],delta['x'],delta['z2']=di,df,dx,dz2 # ditto
    return delta_list      # return lists of error signals for each (unfolded) layer

def doLearningStep(W,xn_list,tn_list,eta,lmbda_by_N=0): # do one BPTT step...
    Z_list = forwardPropagateActivity(xn_list,W);        # forward prop. of input sequence
    delta_list = backPropagateErrors(Z_list, tn_list, W) # get errors by backpropagation through time
    zMD,zMM=np.zeros(shape=W['Wz'].shape,dtype='double'),np.zeros(shape=W['Rz'].shape,dtype='double')
    zKM,zM =np.zeros(shape=W['W21'].shape,dtype='double'),np.zeros(M,dtype='double')
    zK     =np.zeros(K,dtype='double')
    nabla_Wz, nabla_Wi, nabla_Wf, nabla_Wo  = zMD,zMD,zMD,zMD  # initialize gradient matrixes
    nabla_Rz, nabla_Ri, nabla_Rf, nabla_Ro  = zMM,zMM,zMM,zMM  # initialize gradient matrixes
    nabla_pi, nabla_pf, nabla_po, nabla_W21 = zM,zM,zM,zKM     # initialize gradient matrixes/vectors
    nabla_bz, nabla_bi, nabla_bf, nabla_bo, nabla_bz2 = zM,zM,zM,zM,zK  # initialize grad. bias vect.
    for tau in range(len(xn_list)): # loop over shared weights in the enfolded network
        delta=delta_list[tau] # dict of error signals for time step tau
        nabla_Wz = nabla_Wz + np.outer(delta['z'],xn_list[tau]) # compute gradient for Wz
        nabla_Wi = nabla_Wi + np.outer(delta['i'],xn_list[tau]) # compute gradient for Wi
        nabla_Wf = nabla_Wf + np.outer(delta['f'],xn_list[tau]) # compute gradient for Wf
        nabla_Wo = nabla_Wo + np.outer(delta['o'],xn_list[tau]) # compute gradient for Wo
        nabla_bz = nabla_bz + delta['z'] # compute gradient for bz
        nabla_bi = nabla_bi + delta['i'] # compute gradient for bi
        nabla_bf = nabla_bf + delta['f'] # compute gradient for bf
        nabla_bo = nabla_bo + delta['o'] # compute gradient for bo
        nabla_po = nabla_po + np.multiply(delta['o'],Z_list[tau]['c']) # compute gradient for po
        nabla_W21 = nabla_W21 + np.outer(delta['z2'],Z_list[tau]['y']) # compute gradient for Wo
        nabla_bz2 = nabla_bz2 + delta['z2']                            # compute gradient for bz2
        if tau>0:
            nabla_Rz = nabla_Rz + np.outer(delta['z'],Z_list[tau-1]['y']) # compute gradient for Rz
            nabla_Ri = nabla_Ri + np.outer(delta['i'],Z_list[tau-1]['y']) # compute gradient for Ri
            nabla_Rf = nabla_Rf + np.outer(delta['f'],Z_list[tau-1]['y']) # compute gradient for Rf
            nabla_Ro = nabla_Ro + np.outer(delta['o'],Z_list[tau-1]['y']) # compute gradient for Ro
            nabla_pi = nabla_pi + np.multiply(delta['i'],Z_list[tau-1]['c']) # compute gradient for pi
            nabla_pf = nabla_pf + np.multiply(delta['f'],Z_list[tau-1]['c']) # compute gradient for pi
    W_new, nablaED = {}, {}  # initialize new weights and gradients as an empty dicts
    W_new['Wz' ]=W['Wz']*(1.0-lmbda_by_N*eta)-eta*nabla_Wz # weight update, weight decay regularization
    nablaED['Wz' ]=nabla_Wz                               # store also gradient of Wz in dict
    W_new['Wi' ],nablaED['Wi' ]=W['Wi' ]*(1.0-lmbda_by_N*eta)-eta*nabla_Wi , nabla_Wi  # ditto for Wi
    W_new['Wf' ],nablaED['Wf' ]=W['Wf' ]*(1.0-lmbda_by_N*eta)-eta*nabla_Wf , nabla_Wf  # ditto for Wf
    W_new['Wo' ],nablaED['Wo' ]=W['Wo' ]*(1.0-lmbda_by_N*eta)-eta*nabla_Wo , nabla_Wo  # ditto for Wo
    W_new['Rz' ],nablaED['Rz' ]=W['Rz' ]*(1.0-lmbda_by_N*eta)-eta*nabla_Rz , nabla_Rz  # ditto for Rz
```

```
    W_new['Ri' ],nablaED['Ri' ]=W['Ri' ]*(1.0-lmbda_by_N*eta)-eta*nabla_Ri , nabla_Ri  # ditto for Ri
    W_new['Rf' ],nablaED['Rf' ]=W['Rf' ]*(1.0-lmbda_by_N*eta)-eta*nabla_Rf , nabla_Rf  # ditto for Rf
    W_new['Ro' ],nablaED['Ro' ]=W['Ro' ]*(1.0-lmbda_by_N*eta)-eta*nabla_Ro , nabla_Ro  # ditto for Ro
    W_new['bz' ],nablaED['bz' ]=W['bz' ]*(1.0-lmbda_by_N*eta)-eta*nabla_bz , nabla_bz  # ditto for bz
    W_new['bi' ],nablaED['bi' ]=W['bi' ]*(1.0-lmbda_by_N*eta)-eta*nabla_bi , nabla_bi  # ditto for bi
    W_new['bf' ],nablaED['bf' ]=W['bf' ]*(1.0-lmbda_by_N*eta)-eta*nabla_bf , nabla_bf  # ditto for bf
    W_new['bo' ],nablaED['bo' ]=W['bo' ]*(1.0-lmbda_by_N*eta)-eta*nabla_bo , nabla_bo  # ditto for bo
    W_new['pi' ],nablaED['pi' ]=W['pi' ]*(1.0-lmbda_by_N*eta)-eta*nabla_pi , nabla_pi  # ditto for pi
    W_new['pf' ],nablaED['pf' ]=W['pf' ]*(1.0-lmbda_by_N*eta)-eta*nabla_pf , nabla_pf  # ditto for pf
    W_new['po' ],nablaED['po' ]=W['po' ]*(1.0-lmbda_by_N*eta)-eta*nabla_po , nabla_po  # ditto for po
    W_new['W21'],nablaED['W21']=W['W21']*(1.0-lmbda_by_N*eta)-eta*nabla_W21, nabla_W21 # ditto for W21
    W_new['bz2'],nablaED['bz2']=W['bz2']*(1.0-lmbda_by_N*eta)-eta*nabla_bz2, nabla_bz2 # ditto for bz2
    return W_new, nablaED # return new weights (and gradients to verify)

def getError(W,X,T,lmbda=0): # crossentropy error function over data (X,T) for LSTM with output layer
    N,K=len(X),W['W21'].shape[0]             # Number of sequences,output layer size
    E=0.0;                                   # initialize error with 0
    for n in range(N):                       # test all data vectors
        Z_list=forwardPropagateActivity(X[n],W) # get output y for input X[n]
        t_list=T[n]                          # get actual target vector sequence ("one hot" coded)
        e=[-t_list[tau][i]*np.log(Z_list[tau]['z2'][i])\
           for tau in range(len(Z_list)) for i in range(K)\
           if not t_list[tau] is None and t_list[tau][i]>0 and Z_list[tau]['z2'][i]>0] # comp. errors
        E=E+np.sum(e)                        # add sum of component errors to total error
    if lmbda>0:                              # weight decay regularization?
        for w_conn in W.keys():              # loop over all connections matixes and vectors
            EW=np.sum(np.sum(np.multiply(W[w_conn],W[w_conn]))) # sum of squared weights for w_conn
            E=E+0.5*lmbda*EW                 # add weight error
    return E;                                # return final error

def checkDataGradients(W,xn,tn,delta=1e-6,eps=1e-6): # verify partial gradients
    # (i) compute experimental gradients
    W_copy  = {w_conn:np.array(W[w_conn],dtype='double') for w_conn in W.keys()}  # make a copy of W
    W_nabla = {w_conn:np.zeros(shape=W[w_conn].shape,dtype='double') for w_conn in W.keys()}  # init
    for w_conn in W.keys():                  # loop over all connections of the LSTM entwork
        weights_org, weights_copy, nabla = W[w_conn], W_copy[w_conn], W_nabla[w_conn] # get references
        if len(weights_org.shape)==1:        # are weights a weight-vector?
            for i in range(weights_org.shape[0]):        # loop over vector
                weights_copy[i]=weights_org[i]+delta     # compute...
                nabla[i]=getError(W_copy,[xn],[tn],0)    # difference quotient...
                weights_copy[i]=weights_org[i]-delta     # to estimate gradient
                nabla[i]=(nabla[i]-getError(W_copy,[xn],[tn],0))/(2*delta) # store it in nabla
                weights_copy[i]=weights_org[i]           # reset original weights
        elif len(weights_org.shape)==2:      # are weights a weight-matrix?
            for i in range(weights_org.shape[0]):        # loop over matrix rows
                for j in range(weights_org.shape[1]):    # loop over matrix columns
                    weights_copy[i,j]=weights_org[i,j]+delta # compute...
                    nabla[i,j]=getError(W_copy,[xn],[tn],0)  # difference quotient...
                    weights_copy[i,j]=weights_org[i,j]-delta # to estimate gradient
                    nabla[i,j]=(nabla[i,j]-getError(W_copy,[xn],[tn],0))/(2*delta) # store it in nabla
                    weights_copy[i,j]=weights_org[i,j]   # reset to original weights
```

```
        else: assert 0,"UNEXPECTED weights shape!"
    # (iv) get gradients via backpropagation
    W_nablaBP=doLearningStep(W,xn,tn,0,0)[1]
    # (v) evaluate differences between experimental and backprop gradients
    for w_conn in W.keys(): # loop over all connections of the LSTM entwork
        nabla_exp, nabla_bp = W_nabla[w_conn], W_nablaBP[w_conn] # get references to gradients
        len_nabla_exp = np.linalg.norm(nabla_exp.flat)   # Euklid. length of experimental nabla
        diff_nabla = np.linalg.norm((nabla_exp-nabla_bp).flat)   # length of difference vector
        relerr_nabla = diff_nabla/len_nabla_exp          # relative error between exp./bp gradient
        if np.abs(relerr_nabla)>eps or True:
            print("Significant gradient difference for connection w_conn=",w_conn,":")
            print("   len=",len_nabla_exp,"diff=",diff_nabla,"relerr=",relerr_nabla)

# *****************************************************
# Main program
# *****************************************************
if __name__ == '__main__':

    # (i) Create training data: Generate time series being a sum of two sinus functions
    # The learning goal is to predict if values will increase by at least th percents after ph time steps
    N,Tau  =10,100     # number of sequences, length of each sequence
    D,K    =1,3        # dimensions of input vectors, dimensions of outputs (one-hot!)
    X,T=[],[]          # initialize data and targets with empty lists
    f1, f2 =0.05, 0.1  # frequencies of the two sinus functions (in cycles per time step)
    ph =5              # prediction horizon (predict after ph steps)
    th =0.05           # threshold for classifying "increasing" or "decreasing" (e.g., 0.05=5%)
    sig_offset,sig_phase,sig_noise=0.05,5,0.1 # standard deviations for offset, phase, value of the sinus
    t_inc = np.array([1,0,0])  # one-hot codeword for predicting "increasing" value >= v*(1+th)
    t_eq  = np.array([0,1,0])  # one-hot code for predicting "equal" value between v*(1-th) and v*(1+th)
    t_dec = np.array([0,0,1])  # one-hot codeword for predicting "decreasing" value <= v*(1-th)
    for n in range(N):
        offset = np.abs(np.random.normal(0,sig_offset,1)[0]) # offset noise --> added to sinus function
        phase  = np.random.normal(0,sig_phase,1)[0]          # phase noise  --> phase shift for sinus
        tau    = np.array(range(Tau),'float')                # time samples tau=0,...,Tau-1
        x      = np.sin(2*np.pi*f1*tau+phase)+np.sin(2*np.pi*f2*tau+phase)+2.5  # time sequence values
        x      = x+offset+np.abs(np.random.normal(0,sig_noise,Tau)) # add offset and individual noise
        x_list = [np.array([x[tau]]) for tau in range(Tau)]  # x as list of np-arrays
        t_list = [t_eq for tau in range(Tau)] # init sequence of target prediction values with defaults
        for tau in range(0,Tau-ph):
            if x_list[tau+ph][0]>=(x_list[tau][0]*(1.0+th)):t_list[tau]=t_inc   # target "increasing"
            if x_list[tau+ph][0]<=(x_list[tau][0]*(1.0-th)):t_list[tau]=t_dec   # target "decreasing"
        X.append(x_list)       # append new data sequence to data list
        T.append(t_list)       # append new target sequence to target list

    # (ii) Generate LSTM network and train it by Backpropagation Through Time (BPTT)
    M=3             # number of LSTM blocks in hidden layer
    eta0=0.0005     # initial learning rate
    etadecay=500.0  # learning decay: learning rate is eta = eta0/(1+epoch/etadecay) --> 0
    lmbda=0         # regularization coefficient
    nEpochs=50000   # number of training epochs
    fac_init_weight = 1.0 # factor for initializing random weights
```

```
w_def = [['Wz',[M,D]], ['Wi',[M,D]], ['Wf',[M,D]], ['Wo',[M,D]],\
         ['Rz',[M,M]], ['Ri',[M,M]], ['Rf',[M,M]], ['Ro',[M,M]],\
         ['bz',[M]  ], ['bi',[M]  ], ['bf',[M]  ], ['bo',[M]  ],\
         ['pi',[M]  ], ['pf',[M]  ], ['po',[M]  ], \
         ['W21',[K,M]], ['bz2',[K]]] # define names and sizes of weight matrices and vectors
W = {}                               # initialize weight dict
for wd in w_def:                     # loop over all weight definitions
    print("ws1=",wd[1],wd[0])
    if len(wd[1])==1:                # weights vector?
        W[wd[0]] = fac_init_weight*(np.random.rand(wd[1][0])-0.5) # initialize weights randomly
    else:                            # or matrix?
        W[wd[0]] = fac_init_weight*(np.random.rand(wd[1][0],wd[1][1])-0.5) # initialize weights
E=getError(W,X,T,0)            # compute initial data error for the random network...
print("initial data error E=",E)    # and print it
for epoch in range(nEpochs):   # loop over learning epochs
    datc,errc = 0,0            # initialize classification error counters with zero
    for n in range(N):         # loop over all training data
        xn=X[n]                # n-th data sequence
        tn=T[n]                # n-th target sequence
        yn=forwardPropagateActivity(xn,W) # get prediction for sequence xn
        for tau in range(len(xn)-ph):                       # loop over all time steps of yn
            t,y=np.argmax(tn[tau]),np.argmax(yn[tau]['z2']) # target and prediction
            if t!=y: errc=errc+1                    # prediction error? then increase counter
            datc=datc+1                             # count also total number of predictions
        W=doLearningStep(W,xn,tn,eta0/(1.0+epoch/etadecay),lmbda/N)[0] # BPTT
    E=getError(W,X,T,0) # compute data error
    print("after epoch",epoch,"training data error E=",E,"with precision =", datc-errc,"/",datc)
    if epoch%10==0: checkDataGradients(W,X[0],T[0],delta=1e-8,eps=1e-8)
```

7.7 Das Problem der verschwindenden Gradienten

Betrachten Sie nochmal die grundlegenden Gleichungen (5.36)-(5.45) aus Kapitel 5.4, welche das Prädiktions- und Lernverhalten von (azyklischen) neuronalen Netzen mit Backpropagation bestimmen (siehe auch Skizze unten):

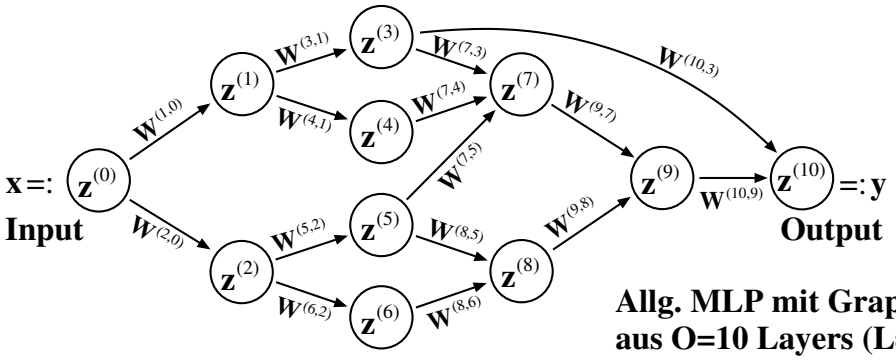
$$\mathbf{a}^{(L)} = \mathbf{b}^{(L)} + \sum_{L' \in \mathrm{PRE}(L)} \mathbf{W}^{(L,L')} \mathbf{z}^{(L')} \qquad \text{mit} \qquad \mathbf{z}^{(L)} = h^{(L)}(\mathbf{a}^{(L)}) \tag{7.44}$$

$$\boldsymbol{\alpha}^{(L)} = \sum_{L' \in \mathrm{SUCC}(L)} \mathbf{W}^{(L',L)^{\mathrm{T}}} \boldsymbol{\delta}^{(L')} \qquad \text{mit} \qquad \boldsymbol{\delta}^{(L)} = \mathbf{J}_{\mathbf{h}}^{(\mathbf{L})^{\mathrm{T}}} \boldsymbol{\alpha}^{(L)} \tag{7.45}$$

$$\Delta \mathbf{W}^{(L,L')} = -\eta \boldsymbol{\delta}^{(L)} \cdot \mathbf{z}^{(L')^{\mathrm{T}}} \qquad \text{und} \qquad \Delta \mathbf{b}^{(L)} = -\eta \boldsymbol{\delta}^{(L)} \tag{7.46}$$

Hier beschreibt (7.44) den Forward-Pass, (7.45) den Backward-Pass, und (7.46) den Gewichtsupdate. Eine Grundvoraussetzung, dass Lernen funktionieren kann ist, dass die Gewichtsänderungen $\Delta \mathbf{W}^{(L',L)}$ und $\Delta \mathbf{b}^{(L)}$ weder unendlich groß noch unendlich klein bzw. Null werden. Wir wollen im folgenden zeigen, dass dies zu verhindern, in Deep Neural Networks (DNN) ein ernstes Problem darstellt. Man spricht hier auch vom Problem der Vanishing Gradients und Exploding Gradients.

Motivierende Frage: Synaptische Gewichte werden typischerweise mit Zufallszahlen initialisiert (z.B. gleichverteilt aus dem Intervall $[-r; r]$; vgl. Anhang A.6). Was passiert mit den Feuerraten $z_j^{(L)}$ bzw. den Fehlersignalen $\delta_j^{(L)}$, wenn alle synaptischen Gewichte sehr klein sind? Und was passiert wenn die meisten synaptischen Gewichte sehr groß sind? Welche Auswirkungen hat das auf das Lernen? Wie könnte man diese Auswirkungen verhindern?



Antwort: Bei kleinen Gewichten (nahe Null) werden wegen der Multiplikation mit $\mathbf{W}^{(L,L')}$ auch die dendritischen Potentiale $a_j^{(L)}$ und Fehlerpotentiale $\alpha_i^{(L')}$ sehr

klein. Dadurch werden typischerweise (je nach Aktivierungsfunktion $h^{(L)}$) auch Feuerraten $z_j^{(L)}$ und Fehlersignale $\delta_i^{(L')}$ sehr klein. Bei Forward-Propagation werden dadurch die $z_j^{(L)}$ für $L = 0, 1, 2, 3, \ldots$ von der Input- in Richtung Output-Layer immer weiter verkleinert. Und bei Back-Propagation werden entsprechend die $\delta_i^{(L')}$ für $L = O, O-1, O-2 \ldots$ von der Output- in Richtung der Input-Layer immer kleiner. Durch die Multiplikation von Feuerraten und Fehlersignalen in (7.46) werden aber dadurch die Gradienten bzw. Gewichtsupdates winzig klein. Das Lernen wird extrem langsam oder hört (bei Unterlauf) sogar ganz auf. Man spricht deshalb von "verschwindenden Gradienten" oder "**Vanishing Gradients**".
Sind dagegen die Gewichte sehr groß, dann passiert genau das Umgekehrte: Die dendritischen Potentiale und Fehlerpotentiale werden immer größer, und damit auch die Feuerraten und Fehlersignale. Dadurch werden die Gradienten extrem groß, und man spricht von "**Exploding Gradients**".

Lösungsansatz: Um Vanishing oder Exploding Gradients zu verhindern muss man offenbar also versuchen die Synapsen (etwa bei der Initialisierung vor dem Lernen) nicht zu stark und nicht zu schwach einzustellen! Um die Problematik und mögliche Lösungen quantitativ zu verstehen wollen wir den **Forward- und Backward Pass "makroskopisch" betrachten**: D.h. im Folgenden abstrahieren wird von den tatsächlichen Gewichten, Aktivierungen und Fehlersignalen der einzelnen Neurone und Synapsen, und betrachten stattdessen in jeder Layer nur noch deren statistischen Verteilungen mit Mittelwerten und Varianzen und analysieren deren wechselseitige Abhängigkeit (siehe Anhang A.4,A.5). Dazu machen wir zunächst die folgenden Annahmen:

i) Alle synaptischen Gewichte $W_{ji}^{(L,L')}$ und Bias-Gewichte $b_k^{(L'')}$ seien unabhängig und (innerhalb einer Layer) ähnlich verteilt, mit identischen Erwartungswerten $E(.)$ und Varianzen Var(.).

ii) Alle synaptischen Gewichte $W_{ji}^{(L,L')}$ haben Mittelwert Null, $E(W_{ji}^{(L,L')}) = 0$.

iii) Alle Gewichte $W_{ji}^{(L,L')}$ und Biase $b_j^{(L)}$ seien unabhängig von den (präsynaptischen) Feuerraten $z_k^{(L')}$.

iv) Alle Gewichte $W_{ji}^{(L,L')}$ seien unabhängig von den (postsynaptischen) Fehlersignalen $\delta_j^{(L)}$.

v) In allen synaptischen Verbindungsmatrizen $\mathbf{W}^{(L,L')}$ ist die Anzahl Synapsen pro Neuron (Fan-In/Fan-Out) identisch (siehe auch folgender Satz).

Unter diesen Annahmen läßt sich die Dynamik der Aktivierungsausbreitung sowie der Fehlersignale im Forward- und Backward-Pass makroskopisch durch die folgenden Gleichungen beschreiben:

Satz 7.4: Makroskopische Dynamik in DNN

Gegeben sei ein azyklisches MLP/DNN mit Layern $L = 0, 1, 2, \ldots, O$, wobei $n^{(L,L')}$ die Anzahl Synapsen ist, die ein Neuron in Layer L' auf ein Neuron in L macht (Fan-In bzw. Fan-Out). Unter obigen Annahmen (i)-(v) läßt sich die Dynamik (7.44),(7.45) der Aktivierungsausbreitung sowie der Fehlersignale im Forward- und Backward-Pass von DNNs durch die folgenden Gleichungen makroskopisch bzw. statistisch beschreiben:

I) **Forward-Pass**:

$$E(a_j^{(L)}) = E(b_j^{(L)}) \tag{7.47}$$

$$\mathrm{Var}(a_j^{(L)}) = \mathrm{Var}(b_j^{(L)}) + \sum_{L' \in \mathrm{PRE}(L)} m_{\mathrm{in}}^{(L,L')} \mathrm{Var}(W_{ji}^{(L,L')}) E([z_i^{(L')}]^2) \tag{7.48}$$

II) **Backward-Pass**:

$$E(\alpha_i^{(L)}) = 0 \tag{7.49}$$

$$\mathrm{Var}(\alpha_i^{(L)}) = \sum_{L' \in \mathrm{SUCC}(L)} m_{\mathrm{out}}^{(L',L)} \mathrm{Var}(W_{ji}^{(L',L)}) E([\delta_j^{(L')}]^2) \tag{7.50}$$

Hierbei sei $m_{\mathrm{in}}^{(L,L')}$ der Layer-spezifische Fan-In eines Neurons in Layer L bzgl. Layer L', d.h. die Anzahl Neurone in Layer L' die eine Synapse $W_{ji}^{(L,L')}$ zu einem Neuron in Layer L haben.
Ähnlich sei $m_{\mathrm{out}}^{(L',L)}$ der Layer-spezifische Fan-Out eines Neurons in Layer L bzgl. Layer L', d.h. die Anzahl Neuron in Layer L' zu denen ein Neuron aus Layer L eine Verbindung hat.

Beweis: I) Ausschreiben der Matrizenmultiplikation in (7.44) ergibt

$$a_j^{(L)} = b_j^{(L)} + \sum_{L'} \sum_i W_{ji}^{(L,L')} z_i^{(L')}$$

und mit der Linearität des Erwartungswerts (A.3) und dem Multiplikationssatz für den Erwartungswert unabhängiger Zufallsvariablen (A.19) folgt wegen $E(W_{ji}^{(L,L')}) = 0$ nach Annahme (ii):

$$E(a_j^{(L)}) = E\left(b_j^{(L)} + \sum_{L',i} W_{ji}^{(L,L')} z_i^{(L')}\right) = E(b_j^{(L)}) + \sum_{L',i} E(W_{ji}^{(L,L')}) E(z_i^{(L')}) = E(b_j^{(L)}) \ .$$

Mit den Summensätzen (A.20),(A.30) der Varianz folgt entsprechend

$$\begin{aligned}
\mathrm{Var}(a_j^{(L)}) &= \mathrm{Var}\left(b_j^{(L)} + \sum_{L',i} W_{ji}^{(L,L')} z_i^{(L')}\right) \overset{(A.20)}{=} \mathrm{Var}(b_j^{(L)}) + \mathrm{Var}(\sum_{L',i} W_{ji}^{(L,L')} z_i^{(L')}) \\
&\overset{(A.30)}{=} \mathrm{Var}(b_j^{(L)}) + \sum_{L',i} \mathrm{Var}(\, W_{ji}^{(L,L')} z_i^{(L')}) + \sum_{L',i} \sum_{\substack{L'',i' \\ \neq L',i}} \mathrm{Cov}\left(\, W_{ji}^{(L,L')} z_i^{(L')},\, W_{ji'}^{(L,L'')} z_{i'}^{(L'')}\right) \\
&\overset{(A.28)}{=} \mathrm{Var}(b_j^{(L)}) + \sum_{L',i} \mathrm{Var}(\, W_{ji}^{(L,L')} z_i^{(L')})
\end{aligned}$$

wobei die letzte Gleichung wegen $\mathrm{Cov}(\, W_{ji}^{(L,L')} z_i^{(L')},\, W_{ji'}^{(L,L'')} z_{i'}^{(L'')}) = 0$ genau wie in Beispiel (A.28) von Seite 220 aus der Unabhängigkeit und Mittelwertfreiheit der Gewichte $W_{ji}^{(L,L')}$ und $W_{ji'}^{(L,L'')}$ nach Annahmen (i) und (ii) folgt. Wegen der Unabhängigkeit von $W_{ji}^{(L,L')}$ und $z_i^{(L')}$) nach Annahme (iii) folgt aus dem Varianz-Multiplikationssatz unabhängiger Zufallsvariablen nach (A.22)

$$\begin{aligned}
&\mathrm{Var}(\, W_{ji}^{(L,L')} z_i^{(L')}) \\
&\quad = \mathrm{Var}(\, W_{ji}^{(L,L')}) \mathrm{Var}(z_i^{(L')}) + E^2(\, W_{ji}^{(L,L')}) \mathrm{Var}(z_i^{(L')}) + \mathrm{Var}(\, W_{ji}^{(L,L')}) E^2(z_i^{(L')}) \\
&\quad = \mathrm{Var}(\, W_{ji}^{(L,L')}) (\mathrm{Var}(z_i^{(L')})) + E^2(z_i^{(L')})) \;=\; \mathrm{Var}(\, W_{ji}^{(L,L')}) E([z_i^{(L')}]^2)
\end{aligned}$$

wobei die letzten beiden Gleichungen aus $E(\, W_{ji}^{(L,L')}) = 0$ nach Annahme (ii) und der Definition der Varianz nach (A.4) folgen. Zusammen ergibt sich also

$$\begin{aligned}
\mathrm{Var}(a_j^{(L)}) &= \mathrm{Var}(b_j^{(L)}) + \sum_{L',i} \mathrm{Var}(\, W_{ji}^{(L,L')}) E([z_i^{(L')}]^2) \\
&= \mathrm{Var}(b_j^{(L)}) + \sum_{L'} m_{\mathrm{in}}^{(L,L')} \mathrm{Var}(\, W_{ji}^{(L,L')}) E([z_i^{(L')}]^2)
\end{aligned}$$

da alle Neuron i einer Layer L nach Annahme (v) denselben Fan-in $m_{\mathrm{in}}^{(L,L')}$ bezüglich Layer L' haben, und alle $W_{ji}^{(L,L')}$ und $z_i^{(L')}$ innerhalb einer Layer (d.h. für alle i) nach Annahme (i) identische Varianzen und Erwartungswerte haben.
II) Eine entsprechende Argumentation ergibt nach Ausschreiben von (7.45)

$$\alpha_i^{(L)} = \sum_{L'} \sum_j W_{ji}^{(L',L)} \delta_j^{(L')}$$

mit den Voraussetzungen (i),(ii),(iv),(v)

$$E(\alpha_i^{(L)}) = E(\sum_{L',j} W_{ji}^{(L',L)} \delta_j^{(L')}) \overset{(A.3,A.19)}{=} \sum_{L',j} E(\, W_{ji}^{(L,L')}) E(\delta_j^{(L')}) \overset{(ii)}{=} 0$$

und

$$
\begin{aligned}
\mathrm{Var}(\alpha_i^{(L)}) &= \mathrm{Var}(\sum_{L',j} W_{ji}^{(L',L)} \delta_j^{(L')}) \\
&\overset{(A.30)}{=} \sum_{L',j} \mathrm{Var}(\, W_{ji}^{(L',L)} \delta_j^{(L')}) + \sum_{L',j} \sum_{\substack{L'',j' \\ \neq L',j}} \mathrm{Cov}\left(W_{ji}^{(L',L)} \delta_j^{(L')},\, W_{j'i}^{(L'',L)} \delta_{j'}^{(L'')} \right) \\
&\overset{(A.28)}{=} \sum_{L',j} \mathrm{Var}(\, W_{ji}^{(L',L)} \delta_j^{(L')}) \\
&\overset{(A.22)}{=} \sum_{L',j} \mathrm{Var}(\, W_{ji}^{(L',L)}) \mathrm{Var}(\delta_j^{(L')}) + E^2(\, W_{ji}^{(L',L)}) \mathrm{Var}(\delta_j^{(L')}) + \mathrm{Var}(\, W_{ji}^{(L',L)}) E^2(\delta_j^{(L')}) \\
&\overset{(ii),(v)}{=} \sum_{L'} m_{\mathrm{out}}^{(L',L)} \mathrm{Var}(\, W_{ji}^{(L',L)}) \left(\mathrm{Var}(\delta_j^{(L')}) + E^2(\delta_j^{(L')}) \right) \\
&\overset{(A.4)}{=} \sum_{L'} m_{\mathrm{out}}^{(L',L)} \mathrm{Var}(\, W_{ji}^{(L',L)}) E([\delta_j^{(L')}]^2) \qquad \square
\end{aligned}
$$

Bemerkungen: Bei Anwendung auf Neuronale Netze mit Backpropagation muss man folgendes bedenken:

1) Die Annahmen (i)-(iv) gelten strikt nur nach Initialisierung der Gewichte und vor dem ersten Gewichtsupdate. Danach ergeben sich strenggenommen sofort (subtile) Abhängigkeiten zwischen Gewichten und Feuerraten oder Fehlersignalen.

2) Annahme (iv) dass Fehlersignale unabhängig von den Fehlersignalen seien, gilt strenggenommen nicht mal vor dem ersten Gewichtsupdate, da die Fehlersignale in der Output-Layer (z.B. $\boldsymbol{\delta}^{(O)} = \mathbf{y} - \mathbf{t}$ nach (5.22,5.23)) typischerweise über die Modell-Outputs $\mathbf{y} := \mathbf{z}^{(O)}$ von allen Gewichten $\mathbf{W}^{(L)}$ abhängen, und sich diese Abhängigkeiten bei der Backpropagation zurück bis zur Input-Layer auf alle Fehlersignale $\boldsymbol{\delta}^{(L)}$ auswirken.

3) Trotz dieser Einschränkungen gilt Satz 7.4 in sehr guter Näherung zumindest zu Beginn des Lernvorgangs, und beschreibt gut das makroskopische Netzwerkverhalten.

4) Da $a_j = \sum_i W_{ji} z_i$ und $\alpha_i = \sum_j W_{ji} \delta_j$ nach (7.44),(7.45) aus vielen Additionen zustandekommen, gilt nach dem Zentralen Grenzwertsatz (siehe Seite 226), dass dendritische Potentiale a_j und Fehlerpotentiale α_i näherungsweise Gauß-verteilt sind. Da Gauß-Verteilungen $\mathcal{N}(x; \mu, \sigma^2)$ nach (A.50) bereits durch Erwartungswert μ und Varianz σ^2 vollständig bestimmt sind, liefert Satz 7.4 eine vollständige statistische Beschreibung der Verteilungen aller a_j und α_i.

5) Nichtlineare Zusammenhänge zwischen Feuerraten und dendritischen Potentialen bzw. zwischen Fehlersignalen und Fehlerpotentialen können bereits für einfache Übertragungsfunktionen vom Typ (5.2) mittels $z_j(a_j) = h(a_j)$ und $\delta_i(\alpha_i) = h'(a_i) \cdot \alpha_i$

für Schwierigkeiten bei der statistischen Analyse sorgen. Immerhin sind die in (7.48) und (7.50) vorkommenden Größen $E([z_i^{(L)}]^2)$ und $E([\delta_j^{(L)}]^2)$ durch die obige Gauß-Annahme abgeleitete Größen der Verteilungen von $z_i^{(L)}$ und $\delta_j^{(L)}$,

$$E([z_j^{(L)}]^2) = E([h^{(L)}(a_j^{(L)})]^2) \overset{(A.38)}{:=} \int_{-\infty}^{\infty} \mathcal{N}(a; E(a_j^{(L)}), \mathrm{Var}(a_j^{(L)}))[h^{(L)}(a)]^2 da \quad (7.51)$$

$$\begin{aligned} E([\delta_i^{(L)}]^2) &= E([h^{(L)\prime}(a_i^{(L)})]^2 \cdot [\delta_i^{(L)}]^2) \overset{(A.19)}{=} E([h^{(L)\prime}(a_i^{(L)})]^2) \cdot E([\alpha_i^{(L)}]^2) \\ &\overset{(A.38)}{:=} \left(\int_{-\infty}^{\infty} \mathcal{N}(a; E(a_i^{(L)}), \mathrm{Var}(a_i^{(L)}))[h^{(L)\prime}(a)]^2 da \right) \\ &\quad \cdot \left(\int_{-\infty}^{\infty} \mathcal{N}(\alpha; E(\alpha_i^{(L)}), \mathrm{Var}(\alpha_i^{(L)}))[\alpha_i^{(L)}]^2 d\alpha \right) \end{aligned} \quad (7.52)$$

wobei wir für (7.52),(A.19),(A.31) zusätzlich annehmen müssen, dass $h^{(L)\prime}(a_i^{(L)})$ und $\alpha_i^{(L)}$ (näherungsweise) unabhängig sind (oder nach (A.31) wenigstens deren Quadrate nicht kovariieren).

Die folgenden Beispiele zeigen wie man mit Hilfe von Satz 7.4 quantitativ abschätzen kann ob die Gradienten verschwinden oder explodieren:

Beispiel 1: Wie lauten die Gleichungen von Satz 7.4 für lineare Neurone mit $h(a) = ca$? Lösung: Für einfache Aktivierungsfunktionen $h : \mathbb{R} \to \mathbb{R}$ mit $h'(a) = c$ gilt

$$z_i^{(L)} = ca_i^{(L)} \qquad \text{und} \qquad \delta_i^{(L)} = h'(a_i^{(L)})\alpha_i^{(L)} = c\alpha_i^{(L)}$$

nach (5.1,7.45,5.40). Damit folgt

$$\begin{aligned} E([z_i^{(L')}]^2) &= E(c^2[a_i^{(L')}]^2) \overset{A.3}{=} c^2 E([a_i^{(L')}]^2) \overset{A.4}{=} c^2(\mathrm{Var}(a_i^{(L')}) + E^2(a_i^{(L')})) \\ &\overset{7.47}{=} c^2 \mathrm{Var}(a_i^{(L')}) + c^2 E^2(b_i^{(L')})) \\ E([\delta_j^{(L')}]^2) &= E(c^2[\alpha_j^{(L')}]^2) \overset{A.3}{=} c^2 E(\alpha_j^{(L')}]^2) \overset{A.4,7.49}{=} c^2 \mathrm{Var}(\alpha_j^{(L')}) \end{aligned}$$

Damit erhält man aus (7.48),(7.50) rekursive Gleichungen für die Varianzen:

$$\mathrm{Var}(a_j^{(L)}) = \mathrm{Var}(b_j^{(L)}) + c^2 \sum_{L' \in \mathrm{PRE}(L)} m_{\mathrm{in}}^{(L,L')} \mathrm{Var}(W_{ji}^{(L,L')})(\mathrm{Var}(a_i^{(L')}) + E^2(b_i^{(L')})) \quad (7.53)$$

$$\mathrm{Var}(\alpha_i^{(L)}) = c^2 \sum_{L' \in \mathrm{SUCC}(L)} m_{\mathrm{out}}^{(L',L)} \mathrm{Var}(W_{ji}^{(L',L)}) \mathrm{Var}(\alpha_j^{(L')}) \quad (7.54)$$

Beispiel 2: Wir betrachten nun ein sequentielles Netzwerk mit $\mathrm{SUCC}(L) := \{L+1\}$ für $L = 0, 1, \ldots, O-1$ aus linearen Neuronen ($z = h(a) = ca$ und $\delta = c\alpha$ wie in Beispiel

1), wobei alle Biase mit Null und alle Gewichte mit Zufallszahlen derselben Varianz σ^2 initialisiert werden (und Mittelwert 0 nach Annahme (ii)). a) Geben Sie rekursive Gleichungen für Mittelwerte und Varianze von Feuerraten und Fehlersignalen an. b) b) Wie entwickeln sich dann die Mittelwerte und Varianze von Feuerraten und Fehlersignalen in Abhängigkeit von der Layer-Tiefe L, falls alle Fan-Ins bzw. Fan-Outs identisch sind? c) Wann tritt das Problem der Vanishing-Gradients auf? Wann Exploding Gradients? Lösung: a) Nach (7.47),(7.49) ist $E(z_j^{(L)}) = cE(a_j^{(L)}) = cE(b_j^{(L)}) = 0$ und $E(\delta_i^{(L)}) = cE(\alpha_i^{(L)}) = 0$. Wegen (A.7) ist $\mathrm{Var}(\delta_i^{(L)}) = \mathrm{Var}(c\alpha_i^{(L)}) = c^2\mathrm{Var}(\alpha)_i^{(L)}$ und ähnlich $\mathrm{Var}(z_j^{(L)}) = c^2\mathrm{Var}(a_j^{(L)}$. Nach Durchmultiplizieren mit c^2 werden (7.53),(7.54) damit zu

$$\mathrm{Var}(z_j^{(L)}) = c^2 m_{\text{in}}^{(L,L-1)}\sigma^2\mathrm{Var}(z_i^{(L-1)}) \tag{7.55}$$

$$\mathrm{Var}(\delta_i^{(L)}) = c^2 m_{\text{out}}^{(L+1,L)}\sigma^2\mathrm{Var}(\delta_j^{(L+1)}) \tag{7.56}$$

b) Falls alle Fan-Ins und Fan-Outs den identischen Wert $m := m_{\text{in}}^{(L,L-1)} = m_{\text{out}}^{(L,L-1)}$ haben (z.B. bei Vollvernetzung mit identischen Layer-Größen) dann erhält man für die Konstante $v := c^2 m\sigma^2$ durch rekursives Ersetzen

$$\mathrm{Var}(z_j^{(L)}) = v\mathrm{Var}(z_j^{(L-1)}) = v^2\mathrm{Var}(z_j^{(L-2)}) = \ldots = v^L\mathrm{Var}(z_j^{(0)}) \tag{7.57}$$

$$\mathrm{Var}(\delta_i^{(L)}) = v\mathrm{Var}(\delta_i^{(L+1)}) = v^2\mathrm{Var}(\delta_i^{(L+2)}) = \ldots = v^{O-L}\mathrm{Var}(\delta_i^{(O)}) \tag{7.58}$$

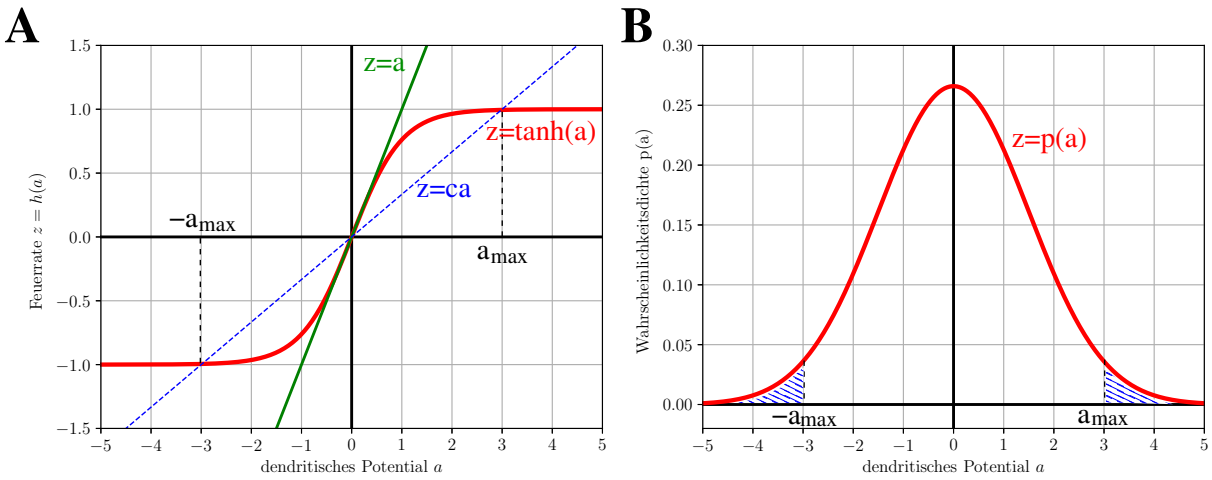
c) Offenbar gilt für tiefe Netzwerke mit Layer-Anzahl $O \to \infty$, dass die Varianzen verschwinden bzw. unendlich groß werden, falls $v < 1$ bzw. $v > 1$ ist. Verschwindende Varianzen $\mathrm{Var}(z_j^{(L)}) \to 0$ und $\mathrm{Var}(\delta_i^{(L)}) \to 0$ bedeuten aber nach (A.5), dass $z_j^{(L)} \to E(z_j^{(L)}) = 0$ und $\delta_i^{(L)} \to E(\delta_i^{(L)}) = 0$. D.h. es tritt das Vanishing Gradients-Problem auf, denn $\frac{\partial E_n}{\partial W_{ji}^{L,L-1}} = \delta_j^{(L)} z_i^{(L-1)} \to 0$. Unendliche große Varianzen $\mathrm{Var}(z_j^{(L)}) \to \infty$ und $\mathrm{Var}(\delta_i^{(L)}) \to \infty$ bedeutet entsprechend $z_j^{(L)} \to \infty$ und $\delta_i^{(L)} \to \infty$ für fast alle Feuerraten und Fehlersignale. [23] D.h. es tritt das Exploding Gradients-Problem auf, denn $\frac{\partial E_n}{\partial W_{ji}^{L,L-1}} = \delta_j^{(L)} z_i^{(L-1)} \to \infty$.

Beispiel 3: Gegeben sei ein lineares sequentielles Netzwerk (wie in Beispiel 2) aus $O = 8$ Layern mit $h(a) = 2a$, wobei jedes Hidden-Neuron zu jeweils $m = 30$ Neuronen der beiden benachbarten Layern verbunden ist. Alle synaptischen Gewichte werden mit

[23] Das gilt zumindest bei hinreichend gutartigen Verteilungen dieser Werte, insbesondere für Gauß-Verteilungen (siehe Skizze auf Seite A.9): Mit zunehmender Varianz $\sigma^2 \to \infty$ wird die Gauß-Glocke immer breiter und flacher, und deshalb wird die Wahrscheinlich, dass ein Zufallswert X kleiner als eine beliebige Konstante C bleibt gegen 0, d.h. $P(|X| < C) = \int_{-C}^{C} \mathcal{N}(x; 0, \sigma^2)dx \to 0$, wobei $\mathcal{N}(x; 0, \sigma^2)$ die Gauß'sche Dichtefunktion von (A.50) ist. Man beachte hierzu, dass wegen des Zentralen Grenzwertsatzes von Seite 226 sowohl $a_j = \sum_i W_{ji} z_i$ als auch $\alpha_i = \sum_j W_{ji}\delta_j$ (Summe von vielen Termen) und damit auch $z_j = ca_j$ und $\delta_i = c\alpha_i$ nahezu Gauß-verteilt sind.

einer Gleichverteilung im Interval $[-r; r]$ initialisiert. Bestimmen Sie die Varianzen der Feuerraten und Fehlersignale in allen Layern L für $\mathrm{Var}(z_j^{(0)}) = 1$ und $\mathrm{Var}(\delta_i^{(O)}) = 1$. a) Für $r = 1$. b) Für $r = 0.01$.
Lösung: Die Gleichverteilung der Gewichte entspricht nach (A.41) einer Varianz von $\sigma^2 = (2r)^2/12 = r^2/3$.
a) Für $r = 1$ ist $\sigma^2 = 1/3$ und $v = c^2 m\sigma^2 = 4 \cdot 30/3 = 40$. D.h. $\mathrm{Var}(z_j^{(1)}) = v\mathrm{Var}(z_j^{(0)}) = 40$, $\mathrm{Var}(z_j^{(2)}) = v40 = 1600$, $\mathrm{Var}(z_j^{(3)}) = v1600 = 64000$, und ähnlich $\mathrm{Var}(z_j^{(4)}), \ldots, \mathrm{Var}(z_j^{(8)}) = 2560000, 1.024e+8, 4.096e+9, 1.64e+11, 6.55e+12$, sowie $\mathrm{Var}(\delta_i^{(7)}) = v\mathrm{Var}(\delta_i^{(8)}) = 40$, $\mathrm{Var}(\delta_i^{(6)}) = v40 = 1600$, $\mathrm{Var}(\delta_i^{(5)}), \ldots, \mathrm{Var}(z_i^{(1)}) = 64000, 2560000, 1.024e+8, 4.096e+9, 1.64e+11$. D.h. Exploding Gradients!
b) Für $r = 0.01$ ist $\sigma^2 = 1/30000$ und $v = 4 \cdot 30/30000 = 4/1000 = 0.004$. D.h. $\mathrm{Var}(z_j^{(1)}), \ldots, \mathrm{Var}(z_j^{(8)}) = 0.004, 1.6e-5, 6.4e-8, 2.56e-2, 2.56e-10, 1.024e-12, 4.096e-15, 1-64e17$ und $\mathrm{Var}(z_j^{(7)}), \ldots, \mathrm{Var}(z_j^{(1)}) = 0.004, 1.6e-5, 6.4e-8, 2.56e-2, 2.56e-10, 1.024e-12, 4.096e-15$. D.h. Vanishing Gradients!

Beispiel 4: Wie könnte man mit Hilfe von Satz 7.4 auch für nichtlineare Aktivierungen $z = h(a)$ auf einfache Weise die makroskopische Dynamik abschätzen, ohne explizit die Integrale in (7.51),(7.52) ausrechnen zu müssen?
Lösung: Je nachdem ob man die Varianzen von a_j und α_i nach oben oder nach unten abschätzen will, kann man $h(a) \approx ca$ durch eine lineare Funktion mit geeigneter Steigung c approximieren (siehe folgende Skizze). Z.B. gilt für $h = \tanh$ dass $h(a) \leq a$ fur alle a, und $h(a) \geq \frac{1}{3}a$ zumindest für $|a| \leq a_{\max} \approx 2.9$ (wie man einfach durch Ausprobieren/Plotten überprüft; siehe Skizze).



Je nach Verteilung der a kann $|a| \leq a_{\max}$ und die resultierenden Abschätzungen mit

sehr hoher Wahrscheinlichkeit P_{conf} erfüllt sein, z.B. für Gauß-Verteilung mit $E(a) = 0$ und $\text{Var}(a) = \sigma^2$ gilt nach (A.53) auf Seite 228

$$P_{\text{conf}} := P(|a| \leq a_{\max}) = \int_{-a_{\max}}^{a_{\max}} \mathcal{N}(a; 0, \sigma^2)\, da \overset{(A.53)}{=} \text{erf}\left(\frac{a_{\max}}{\sigma}\right) .$$

Konkreter kann man fordern, dass (im Mittel) weniger als ein Neuron die Bedingung $|a| \leq a_{\max}$ verletzt, d.h. $M(1 - P_{\text{conf}}) \overset{!}{\leq} \epsilon$, wobei M die Gesamtzahl der relevanten Neurone [24] und ϵ die erlaubte Schranke sei (für $0 < \epsilon < 1$). Einsetzen von P_{conf} ergibt $1-\text{erf}\left(\frac{a_{\max}}{\sigma}\right) \overset{!}{\leq} \frac{\epsilon}{M}$ und mit $\text{erfc}(a) := 1-\text{erf}(a)$ aus (A.54) nach beideitigem Anwenden von erfc^{-1}

$$\text{erfc}\left(\frac{a_{\max}}{\sigma}\right) \overset{!}{\leq} \frac{\epsilon}{M} \qquad \text{bzw.} \quad a_{\max} \overset{!}{\geq} \sigma\, \text{erfc}^{-1}\left(\frac{\epsilon}{M}\right)$$

. Für kleinstmögliches $a_{\max}$ folgt aus $h(a_{\max}) = c a_{\max}$ also (nun wieder mit Layer-Index L)

$$c^{(L)} = \frac{h(a_{\max}^{(L)})}{a_{\max}^{(L)}} \qquad \text{für} \quad a_{\max}^{(L)} := \sigma^{(L)}\, \text{erfc}^{-1}\left(\frac{\epsilon}{M^{(L)}}\right)$$

was eine sinnvolle lineare untere Schranke $h(a) \tilde{\geq} c^{(L)} a$ definiert, welche für praktisch alle Neurone von Layer L erfüllt ist. Wir fassen dieses Zwischenergebnis im folgenden Lemma zusammen:

Lemma 7.5: Lineare Abschätzung von Sigmoid-Funktionen

Sei $h(a)$ eine sigmoide Aktivierungsfunktion (z.B. $h = \tanh$) mit folgenden Eigenschaften: 1) stetig und streng monoton steigend; 2) $h(0) = 0$ und punktsymmetrisch mit $h(-a) = -h(a)$; 3) Wendepunkt im Ursprung bzw. konvex für $a > 0$. Dann gelten für die denritischen Potentiale a der Neurone aus Layer L die Abschätzungen:

$$|h(a)| \leq h'(0)|a| \tag{7.59}$$

$$|h(a)| \tilde{\geq} c^{(L)}|a| \quad \text{für} \quad c^{(L)} := \frac{h(a_{\max}^{(L)})}{a_{\max}^{(L)}} \quad \text{mit} \quad a_{\max}^{(L)} := \sigma^{(L)}\, \text{erfc}^{-1}\left(\frac{\epsilon}{M^{(L)}}\right) \tag{7.60}$$

wobei in der zweiten Abschätzung angenommen wird, dass die dendritischen Potentiale a Gauß-verteilt sind mit $E(a) = 0$ und $\text{Var}(a) = {\sigma^{(L)}}^2$, und die Abschätzung $\tilde{\geq}$ bedeutet "fast immer $\geq$" bis auf einen Anteil $\epsilon M^{(L)}$ der $M^{(L)}$ Neurone in Layer L. Siehe auch Fußnote 24 für eine alternative Abschätzung.

[24] Relevante Neurone sind z.B. die Neurone in der betrachteten Layer. Alternativ könnte man auch fordern, dass die Wahrscheinlichkeit, dass irgendein Neuron diese Bedingung verletzt kleiner als eine vorgegebene Schranke ϵ ist, $1-P_{\text{conf}}^M \overset{!}{<} \epsilon$. Dies führt analog auf die Bedingung $a_{\max} \overset{!}{\geq} \sigma\, \text{erf}^{-1}\left((1-\epsilon)^{1/M}\right)$, wie man leicht nachrechnen kann.

Damit kann man die Integrale in (7.52) wie folgt abschätzen:

$$c^{(L)} \cdot {\sigma^{(L)}}^2 \tilde{\leq} E([h^{(L)}(a_j^{(L)})]^2) \leq [{h^{(L)}}'(0)]^2 \cdot {\sigma^{(L)}}^2 \tag{7.61}$$

$$1 - 2{\sigma^{(L)}}^2 + 3{c^{(L)}}^4 {\sigma^{(L)}}^4 \tilde{\leq} E([{h^{(L)}}'(a_i^{(L)})]^2) \tilde{\leq} 1 - 2{c^{(L)}}^2 {\sigma^{(L)}}^2 + 3{\sigma^{(L)}}^4 \tag{7.62}$$

wobei (7.62) nur für $h = \tanh$ gilt.

Beweis: (7.59) und (7.60) wurden bereits oben hergeleitet. Damit folgt unmittelbar auch (7.61), z.B. für die obere Schranke $E([h(a)]^2) = \int p(a)h(a)^2 da = \int p(a)|h(a)|^2 da \leq \int p(a)(h'(0))^2|a|^2 da = (h'(0))^2 E(a^2) = (h'(0))^2 \mathrm{Var}(a)$. Die untere Schranke folgt analog. Um (7.62) für $h = \tanh$ zu zeigen verwenden wir $h'(a) = 1 - h^2(a)$ nach (5.26), sodass mit $[h'(a)]^2 = [1 - h^2(a)]^2 = 1 - 2h^2(a) + h^4(a)$ und $h'(0) = 1$ aus (7.59),(7.60) sich die Abschätzungen

$$\begin{aligned} E([h'(a)]^2) &= \int p(a)(1 - 2h^2(a) + h^4(a))da \\ &\tilde{\leq} \int p(a)(1 - 2c^2a^2 + a^4)da = 1 - 2c^2E(a^2) + E(a^4) \\ E([h'(a)]^2) &= \int p(a)(1 - 2h^2(a) + h^4(a))da \\ &\tilde{\geq} \int p(a)(1 - 2a^2 + c^4a^4)da = 1 - 2E(a^2) + c^4E(a^4) \end{aligned}$$

ergeben. Damit folgt (7.62) unmittelbar aus den 2. und 4. Momenten $E(a^2) = \sigma^2$ und $E(a^4) = 3\sigma^4$ der Gauß-Verteilung. [25] □

Beispiel 5: Was wäre eine allgemeine Strategie um sowohl Vanishing als auch Exploding Gradients (zumindest zu Beginn des Lernvorgangs) auszuschließen?
Lösung: Man kann versuchen die Gewichte und Biase so zu initialisieren, dass die Verteilungs-Parameter (Mittelwerte und Varianzen) der dendritischen Potentiale $a_j^{(L)}$ und der Fehlerpotentiale $\alpha_i^{(L)}$ definierte Werte annehmen (z.B. alle Mittelwerte Null und alle Varianzen σ^2). Dann definiert Satz 7.4 ein (i.Allg. nichtlineares) Gleichungssystem, mit den Variablen $E(b_j^{(L)})$, $\mathrm{Var}(b_j^{(L)})$, $\mathrm{Var}(W_{ji}^{(L,L')})$. Falls das Netzwerk aus O Layern und V Verbindungen besteht, dann gibt es also offenbar $2O + V$ Variablen und $2O$ Gleichungen. Da es mehr Variablen als Gleichungen gibt, sollten geeignete Lösungen existieren, die (aufgrund der Nichtlinearität) numerisch bestimmt werden können.

Unter vereinfachenden Annahmen werden häufig die Biase mit Null initialisiert, d.h. $E(b_j^{(L)}) = 0$ und $\mathrm{Var}(b_j^{(L)}) = 0$. In diesem Fall gibt es noch V Variablen und $2 \cdot O$ Gleichungen. Damit sollten geeigente Lösungen zumindest noch dann existieren, falls

[25] Wir zeigen in den Übungen, dass für Gauß-verteile Zufallsvariablen a mit $E(a) =: \mu$ und $\mathrm{Var}(a) =: \sigma^2$ gilt, dass $E(a^4) = \int_{-\infty}^{\infty} a^4 \mathcal{N}(a; \mu, \sigma^2) = \mu^4 + 6\mu^2\sigma^2 + 3\sigma^4$ gilt. Deshalb $E(a^4) = 3\sigma^4$ für $\mu = E(a) = 0$.

das Netzwerk mindestens doppelt soviele Verbindungen wie Layers hat. Andernfalls kann man versuchen durch geignete Heuristiken Näherungslösungen zu finden (siehe folgendes Kapitel).

7.8 Gewichtsinitialisierung nach Xavier und He

Beispiel 5 von Seite 194 diskutiert eine allgemeine Strategie, um gewünschte Verteilungen der dendritischen Potentiale $a_j^{(L)}$ und Fehlerpotentiale $\alpha_i^{(L)}$ zu erhalten, die weder verschwinden noch explodieren. Im Folgenden geben wir Lösungen für zwei wichtige Spezialfälle an:

Xavier-Initialisierung: [26] Wir nehmen lineare Aktivierungsfunktionen $h(a) = a$ oder *quasi-lineare* Aktivierungsfunktionen wie etwa [27]

$$h(a) = \tanh(a) = \tanh(0) + \tanh'(0)a + O(a^2) \approx a \qquad \text{für kleine } |a| \ll 1 \tag{7.63}$$

an. Diesen Fall haben wir in allgemeiner Form bereits in Beispiel 1 auf Seite 190 behandelt. Wir machen gegenüber diesem Beispiel (und Satz 7.4) zwei zusätzliche Annahmen:

- Alle Biase werden mit konstant Null initialisiert, d.h. $E(b_i^{(L)}) = 0$ und $\mathrm{Var}(b_i^{(L)}) = 0$ für alle Layer L.
- Um Vanishing oder Exploding Gradients zu vermeiden sollen am Ende alle dendritischen Potentiale bzw. Fehlerpotentiale dieselbe endliche Varianz haben, d.h. $\mathrm{Var}(a_j^{(L)}) = \mathrm{Var}(a_j) =: \sigma_a^2$ bzw. $\mathrm{Var}(\alpha_i^{(L)}) = \mathrm{Var}(\alpha_i) =: \sigma_\alpha^2$ für alle L.

Damit vereinfachen sich (7.53) und (7.54) für $h(a) = ca$ mit $c = 1$ und nach Ausklammern der vom Summationsindex L' unabhängigen Terme zu den Bedingungen

$$\mathrm{Var}(a_j) \stackrel{!}{=} \mathrm{Var}(a_i) \sum_{L' \in \mathrm{PRE}(L)} m_{\mathrm{in}}^{(L,L')} \mathrm{Var}(W_{ji}^{(L,L')}) \qquad \text{für alle } L \tag{7.64}$$

$$\mathrm{Var}(\alpha_i) \stackrel{!}{=} \mathrm{Var}(\alpha_j) \sum_{L' \in \mathrm{SUCC}(L)} m_{\mathrm{out}}^{(L',L)} \mathrm{Var}(W_{ji}^{(L',L)}) \qquad \text{für alle } L \tag{7.65}$$

[26] Siehe: Xavier Glorot, Yoshua Bengio: Understanding the difficulty of training deep feedforward neural networks, Proceedings of the 13th International Conference on Artificial Intelligence and Statistics, pp249-256, 2010. Der Name **Xavier-Initialisierung** kommt vom Vornamen des Erstautors. Manchmal sagt man auch **Glorot-Initialisierung** nach seinem Nachnamen.

[27] Nach (5.25) und (5.26) ist $\tanh(0) = 0$ und $\tanh'(0) = 1 - \tanh^2(0) = 1$, und damit ist die Tangente (oder Tayler-Entwicklung) im Ursprung die identische Abbildung $h(a) \approx h(0) + h'(0)a = a$. Siehe auch die Skizze auf Seite 192.

und nach Kürzen zu

$$\sum_{L' \in \mathrm{PRE}(L)} m_{\mathrm{in}}^{(L,L')} \mathrm{Var}(W_{ji}^{(L,L')}) \stackrel{!}{=} 1 \qquad \text{für alle } L \tag{7.66}$$

$$\sum_{L' \in \mathrm{SUCC}(L)} m_{\mathrm{out}}^{(L',L)} \mathrm{Var}(W_{ji}^{(L',L)}) \stackrel{!}{=} 1 \qquad \text{für alle } L \tag{7.67}$$

d.h. ein lineares Gleichungssystem aus $2 \cdot O$ Gleichungen und V Unbekannten $\mathrm{Var}(W_{ji}^{(L,L')})$, wobei V die Anzahl der zu initialisierenden Verbindungen $\mathbf{W}^{(L,L')}$ ist. Im Allgemeinen existiert dann eine Lösung falls es mindestens soviele Unbekannte wie Gleichungen gibt, d.h. für $V \geq 2 \cdot O$.

Ist dies nicht erfüllt (z.B. für sequentielle Netzwerke gilt $V = O < 2 \cdot O$) kann man (7.66) und (7.67) als zwei getrennte lineare Gleichungssysteme (für Forward- und Backward-Pass) betrachten. Jedes Gleichungssystem hat O Gleichungen und $V \geq O$ Unbekannten, d.h. für jedes der beiden Gleichungssysteme existiert eine eigene Lösung für $\mathrm{Var}(W_{ji}^{(L,L')})$. In der Praxis mittelt man dann einfach über die beiden Lösungen.

Tatsächlich nimmt die **Xavier-Initialisierung in der Praxis** eine sequentielle Netzwerkstruktur an, d.h. $\mathrm{PRE}(L) = \{L-1\}$ und $\mathrm{SUCC}(L) = \{L+1\}$. Mit der vereinfachten Notation für Fan-In $m_{\mathrm{in}}^{(L)} := m_{\mathrm{in}}^{(L,L-1)}$, Fan-Out $m_{\mathrm{out}}^{(L)} := m_{\mathrm{out}}^{(L,L-1)}$, und Verbindungen $\mathbf{W}^{(L)} := \mathbf{W}^{(L,L-1)}$ vereinfachen sich (7.66) und (7.67) dann zu

$$m_{\mathrm{in}}^{(L)} \mathrm{Var}(W_{ji}^{(L)}) \stackrel{!}{=} 1 \qquad \text{und} \qquad m_{\mathrm{out}}^{(L)} \mathrm{Var}(W_{ji}^{(L)}) \stackrel{!}{=} 1 \tag{7.68}$$

Da man offenbar nicht beide Bedingungen gleichzeitig erfüllen kann (außer Fan-In und Fan-Out stimmen zufällig überein) ersetzt die Xavier-Initialisierung Fan-In und Fan-Out einfach durch deren Mittelwert $\frac{m_{\mathrm{in}}^{(L)} + m_{\mathrm{out}}^{(L)}}{2}$ um die beiden Bedingungen (7.68) zu verschmelzen:

Satz 7.6: Xavier-Initialisierung

Um die Gefahr von Vanishing Gradients oder Exploding Gradients zu minimieren, initialisiert man sequentielle Neuronale Netzwerke mit linearer oder tanh Aktivierung, indem man alle Biase auf Null setzt, $b_j^{(L)} = 0$, und die synaptischen Gewichte jeder Verbindung $\mathbf{W}^{(L)}$ mit Zufallszahlen initialisert, sodass

$$E(W_{ji}^{(L)}) = 0 \qquad \text{und} \qquad \mathrm{Var}(W_{ji}^{(L)}) = \frac{2}{m_{\mathrm{in}}^{(L)} + m_{\mathrm{out}}^{(L)}} \tag{7.69}$$

gilt, wobei $m_{\mathrm{in}}^{(L)}$ und $m_{\mathrm{out}}^{(L)}$ Fan-In und Fan-Out der Verbindung sind.

D.h. man kann die synaptischen Gewichte $W_{ji}^{(L)}$ z.B. mit gleichverteilten Zufallszahlen aus dem Intervall $[-r; r]$ oder mit Gauß-verteilten Zufallszahlen mit Erwar-

tungswert $\mu = 0$ und Varianz σ^2 initialisieren wobei für die Verteilungs-Parameter gilt (siehe Anhang A.6,A.8):

$$r = \sqrt{\frac{6}{m_{\text{in}}^{(L)} + m_{\text{out}}^{(L)}}} \tag{7.70}$$

$$\sigma = \sqrt{\frac{2}{m_{\text{in}}^{(L)} + m_{\text{out}}^{(L)}}} \tag{7.71}$$

Beweis: (7.69) folgt unmittelbar aus (7.68) durch die dort beschriebene Mittelungs-Heuristik. Eine Gleichverteilung aus dem Intervall $[-r; r]$ hat nach (A.39) den Erwartungswert $E(W_{ji}^{(L)}) = 0$ und nach (A.41) die Varianz $\text{Var}(W_{ji}^{(L)}) = \frac{(r-(-r))^2}{12} = \frac{4r^2}{12} = \frac{r^2}{3}$. Gleichsetzen mit (7.69) ergibt $\frac{r^2}{3} = \frac{2}{m_{\text{in}}^{(L)}+m_{\text{out}}^{(L)}}$ und nach Auflösen folgt (7.70). Entsprechend hat die Gauß-Verteilung Varianz σ^2 und Gleichsetzen ergibt $\sigma^2 = \frac{2}{m_{\text{in}}^{(L)}+m_{\text{out}}^{(L)}}$ bzw. (7.71). □

Bemerkungen:

- Man nennt die Xavier-Initialisierung oft auch Glorot-Initialisierung, siehe Fußnote 26 auf Seite 195. Sie ist die Standard-Initialisierung bei vielen Machine-Learning-Bibliotheken wie etwa Keras.
- Für nichtlineare Aktivierungen wie $h = \tanh$ ist die Linearitätsannahme (7.63) für große $m_{\text{in}}^{(L)}, m_{\text{out}}^{(L)} \gg 1$ hinreichend gut erfüllt, da wegen (7.69) die meisten Gewichte mit sehr kleinen Werten initialisiert werden, sodass die dendritischen Potentiale und Fehlerpotentiale nahe Null sind, wo tanh beinahe linear verläuft (siehe Skizze auf Seite 192). Offenbar kann $\sigma_a^2 := \text{Var}(a_j)$ beliebig (klein) gewählt werden (z.B. indem die Input-Vektoren skaliert werden).
- Wie oben begründet ist die Xavier-Initialisierung nicht perfekt, sondern verhindert Exploding und Vanishing Gradients nur in sequentiellen Netzwerken, falls die Fan-Ins und Fan-Outs sich nicht zu sehr unterscheiden. Trotzdem verwendet man in der Praxis häufig Xavier-Initialisierung auch für nicht-sequentielle Netzwerke und/oder bei stark unterschiedlichen Fan-Ins und Fan-Outs.

Beispiele: (Details und Lösungen siehe Übungen)

a) Betrachte ein sequentielles Netzwerk aus $O = 8$ vollverbundenenen Layern $L = 0, 1, \ldots, O$ der Größen $M^{(0)}, \ldots, M^{(8)} = 500, 200, 200, 200, 200, 100, 100, 100, 10$ mit tanh-Aktivierungen in allen Layern $L = 1, 2, \ldots, O$. Bestimme für jede synaptische Verbindung $\mathbf{W}^{(L,L-1)}$ die Varianzen für Xavier-Initialisierung. Welche Parameter r bzw. σ ergeben sich daraus?

b) Rechnen Sie mit Hilfe von (7.64) die Varianzen der dendritischen Potentiale in jeder Layer aus. Sie können annehmen, dass die Inputs $a^{(0)}$ standardnormalverteilt sind.

c) Rechnen Sie mit Hilfe von (7.65) die Varianzen der Fehlerpotentiale in jeder Layer aus.

d) Implementieren Sie ein Netzwerk in Python und bestimmen Sie die Varianzen experimentell. Warum stimmen die theoretischen Werte nicht perfekt?

e) Betrachten Sie nun das Netzwerk von (a) wo es zusätzliche "Skip-Verbindungen" gibt, die immer eine Layer überspringen, d.h. Verbindungen vom Typ $\mathbf{W}^{(L,L-2)}$ enthalten. AuSSerdem sei noch eine Verbindung $\mathbf{W}^{(3,0)}$ enthalten, die zwei Layer überspringt, sodass genau $V = 2 \cdot O$ gilt. Vergleichen Sie Xavier-Initialisierung nach Satz 7.6 mit der Initialisierung durch Lösung des Linearen Gleichungssystems (7.66),(7.67). Überprüfen und vergleichen Sie die Ergebnisse mit experimentellen Netzwerk-Implementierungen ähnlich wie in (d).

f) Betrachten Sie nun eine Variante von (a) bei der die Bias-Varianzen $\mathrm{Var}(b_j^{(L)})$ zusätzliche Freiheitsgrade darstellen. Stellen Sie mit Hilfe von Satz 7.4 ein lineares Gleichungssystem für die Varianzen der Gewichte und Biase auf. Überprüfen Sie die Lösungen experimentell ähnlich wie in (d) und (e).

He-Initialisierung: [28] Während die Xavier-Initialisierung quasi-lineare Aktivierungen nach (7.63) annimmt, wollen wir hier diesen Ansatz verallgemeinern, indem wir stückweise lineare Aktivierungsfunktionen der Art

$$h(a) = \begin{cases} ca + u & , a \geq 0 \\ da + u & , a < 0 \end{cases} \tag{7.72}$$

betrachten (siehe folgende Skizze). Man beachte dass man damit wie (7.63) verschiedene häufig verwendete Aktivierungsfunktionen darstellen oder approximieren kann:

- Tangenshyperbolicus: $\tanh(a) \approx a = h(a)$ für $c = d = 1$ und $u = 0$.
- Logistische Sigmoide von Lemma **??** auf Seite **??**: $\sigma(a) \approx 0.5 + 0.25a = h(a)$ für $c = d = 0.25$ und $u = 0.5$.
- Rectified Linear Unit (ReLU): $\mathrm{relu}(a) := \mathrm{ramp}(a) = h(a)$ für $d = u = 0$ und $c = 1$.
- Parameterized oder Leaky Rectified Linear Unit (PReLU, LReLU): $\mathrm{prelu}(a) = \mathrm{lrelu}(a) = h(a)$ für $u = 0$.

[28] Siehe: Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun: Delving deep into rectifiers: surpassing human-level performance on ImageNet classification. Proceedings of the IEEE International Conference on Computer Vision (ICCV), 2015. Der Name **He-Initialisierung** kommt vom Vornamen des Erstautors. Manchmal sagt man auch **Kaiming-Initialisierung** nach seinem Nachnamen.

Um die makroskopische Dynamik eines neuronalen Netzes für (7.72) zu beschreiben können wir wieder den allgemeinen Satz 7.4 anwenden: Hierfür berechnen wir für $z = h(a)$ und achsensymmetrische (z.B. Gauß'sche) Dichte $p(a)$ mit $p(-a) = p(a)$ und $E(a) = 0$ ähnlich wie in (7.51) nach Weglassen der Layer/Neuron-Indexe

$$\begin{aligned}
E(z^2) &\overset{(A.38)}{=} \int_{-\infty}^{\infty} p(a)h^2(a)da = \int_{-\infty}^{0} p(a)(da+u)^2 da + \int_{0}^{\infty} p(a)(ca+u)^2 da \\
&= \int_{0}^{\infty} p(a)(ca+u)^2 da - \int_{0}^{-\infty} p(a)(da+u)^2 da \\
&\overset{(a'=-a;da'/da=-1)}{=} \int_{0}^{\infty} p(a)(ca+u)^2 da - \int_{0}^{\infty} p(-a')(-da'+u)^2(-da') \\
&\overset{(p(-a)=-p(a))}{=} \int_{0}^{\infty} p(a)[c^2a^2 + 2cua + u^2 + (d^2a^2 - 2dua + u^2)]da \\
&= \int_{0}^{\infty} p(a)[(c^2+d^2)a^2 + 2(c-d)ua + 2u^2]da \\
&= (c^2+d^2)\int_{0}^{\infty} p(a)a^2 + 2(c-d)u\int_{0}^{\infty} p(a)ada + 2u^2\int_{0}^{\infty} p(a) \\
&\overset{(p(-a)=-p(a))}{=} \frac{c^2+d^2}{2}\int_{-\infty}^{\infty} p(a)a^2 + (c-d)u\int_{-\infty}^{\infty} p(a)|a|da + 2u^2\frac{1}{2} \\
&\overset{(A.38)}{=} \frac{c^2+d^2}{2}E(a^2) + (c-d)uE(|a|) + u^2
\end{aligned} \tag{7.73}$$

Falls a Gauß-verteilt mit $E(a) = 0$ und $\mathrm{Var}(a) = \sigma^2$ dann ist $E(|a|) = \sqrt{\frac{2}{\pi}}\sigma$ und es folgt [29]

$$E(z^2) = \frac{c^2+d^2}{2}\mathrm{Var}(a) + (c-d)u\sqrt{\frac{2}{\pi}\mathrm{Var}(a)} + u^2 \tag{7.74}$$

Entsprechend erhalten wir für $\delta = h'(a) \cdot \alpha$ ähnlich wie in (7.52)

$$\begin{aligned}
E(\delta^2) \tilde{=} E(h'^2(a)) \cdot E(\alpha^2) &= E(\alpha^2)\int_{-\infty}^{\infty} p(a)h'^2(a)da \\
&= \mathrm{Var}(\alpha) \cdot \left(\int_{-\infty}^{0} p(a)d^2 da + \int_{0}^{\infty} p(a)c^2 da\right) \\
&= \mathrm{Var}(\alpha) \cdot \left(\frac{d^2}{2} + \frac{c^2}{2}\right) = \frac{c^2+d^2}{2}\mathrm{Var}(\alpha)
\end{aligned} \tag{7.75}$$

[29] Für Gauß'sche Dichte $p(a) = \frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{a^2}{2\sigma^2}}$ folgt mit der Substitution $y = -\frac{a^2}{2\sigma^2}$ bzw. $\frac{dy}{da} = -\frac{a}{\sigma^2}$ oder $da = -dy\frac{\sigma^2}{a}$ der Erwartungswert $E(|a|) = \int_{-\infty}^{\infty} p(a)|a|da = 2\int_{0}^{\infty} p(a)ada = 2\frac{1}{\sqrt{2\pi}\sigma}\int_{0}^{\infty} e^{-\frac{a^2}{2\sigma^2}}ada$ $= \sqrt{\frac{2}{\pi}}\frac{1}{\sigma}\int_{0}^{-\infty} e^y a\frac{\sigma^2}{a}(-dy) = \sqrt{\frac{2}{\pi}}\sigma\int_{-\infty}^{0} e^y dy = \sqrt{\frac{2}{\pi}}\sigma \cdot (1-0) = \sqrt{\frac{2}{\pi}}\sigma$.

Um Vanishing und Exploding Gradients zu verhindern, fordert man nun wieder, dass die Varianzen der dendritischen Potentiale und Fehlerpotentiale unabhängig von der Layer gleich bleiben, $\mathrm{Var}(a_i^{(L)}) =: \mathrm{Var}(a_i) =: \sigma_a^2$ und $\mathrm{Var}(\alpha_j^{(L)}) =: \mathrm{Var}(\alpha_j) =: \sigma_\alpha^2$. Dann ergibt sich nach Einsetzen von (7.74) und (7.75) in Satz 7.4 für $E(b_j^{(0)}) = 0$ wieder ein lineares Gleichungssystem, womit das folgende Lemma gezeigt ist:

Lemma 7.7: Verallgemeinerte He-Initialisierung

Gegeben sei ein Neuronales Netzwerk mit stückweise linearen Aktivierungsfunktionen nach (7.72). Damit die Varianzen σ_a^2 und σ_α^2 von dendritischen Potentialen und Fehlerpotentialen in allen Layern konstant bleiben (und damit keine Vanishing/Exploding Gradients auftreten), muss man Biase und Gewichte mit erwartungswertfreien Zufallszahlen initialisieren ($E(b_j^{(L)}) = E(W_{ji}^{(L)}) = 0$), wobei die Varianzen $\mathrm{Var}(b_j^{(L)})$ und $\mathrm{Var}(W_{ji}^{(L',L)})$ Lösung des folgenden linearen Gleichungssystems (LGS) sein müssen:

$$\sigma_a^2 = \mathrm{Var}(b_j^{(L)}) + \left(\frac{c^2+d^2}{2}\sigma_a^2 + (c-d)u\sigma_a + u^2\right) \cdot \sum_{L' \in \mathrm{PRE}(L)} m_{\mathrm{in}}^{(L,L')} \mathrm{Var}(W_{ji}^{(L,L')}) \tag{7.76}$$

$$1 \cdot \cancel{\sigma_\alpha^2} = \frac{c^2+d^2}{2}\cancel{\sigma_\alpha^2} \sum_{L' \in \mathrm{SUCC}(L)} m_{\mathrm{out}}^{(L',L)} \mathrm{Var}(W_{ji}^{(L',L)}) \tag{7.77}$$

Das LGS besteht aus $2 \cdot O$ Gleichungen und $O + V$ Unbekannten, und besitzt wegen $V \geq O$ im Allg. eine Lösung.

Bemerkungen:

- Man muss darauf achten, dass die Lösungen des LGS positive Werte sind, denn Varianzen können nicht negativ sein!
- Man kann vereinfachend wieder sequentielle Netzwerke betrachten: Mit den Notationen von Satz 7.6 werden (7.76) und (7.77) zu

$$\sigma_a^2 = \mathrm{Var}(b_j^{(L)}) + \left(\frac{c^2+d^2}{2}\sigma_a^2 + (c-d)u\sigma_a + u^2\right) \cdot m_{\mathrm{in}}^{(L)} \mathrm{Var}(W_{ji}^{(L)}) \tag{7.78}$$

$$1 = \frac{c^2+d^2}{2} m_{\mathrm{out}}^{(L)} \mathrm{Var}(W_{ji}^{(L)}) \tag{7.79}$$

  Es ist leicht hier eine explizite Lösung des LGS zu finden: Aus (7.79) folgt $\mathrm{Var}(W_{ji}^{(L)}) = \frac{2}{(c^2+d^2)m_{\mathrm{out}}^{(L)}}$ und Einsetzen in (7.78) liefert $\mathrm{Var}(b_j^{(L)})$. Um zu verhindern dass $\mathrm{Var}(b_j^{(L)}) < 0$ passiert, kann man eventuell die Offests u manipulieren. Details siehe Übungen.

- Damit Lösungen immer garantiert sind kann man (7.78) und (7.79) ähnlich wie für Satz 7.6 als getrennte Gleichungssysteme auffassen und diese separat lösen,

$$\mathrm{Var}(W_{ji}^{(L)}) = \frac{\sigma_a^2 - \mathrm{Var}(b_j^{(L)})}{\left(\frac{c^2+d^2}{2}\sigma_a^2 + (c-d)u\sigma_a + u^2\right) \cdot m_{\text{in}}^{(L)}} \tag{7.80}$$

$$\mathrm{Var}(W_{ji}^{(L)}) = \frac{2}{(c^2+d^2)m_{\text{out}}^{(L)}} \tag{7.81}$$

  und ähnlich wie in Satz 7.6 über deren Lösungen mitteln.

- Für **lineare Aktivierung** $h(a) = a$ mit $c = d = 1$, $u = 0$ und $\mathrm{Var}(b_j^{(L)}) = 0$ erhält man dann wieder $\mathrm{Var}(W_{ji}^{(L)}) = \frac{1}{m_{\text{in}}^{(L)}}$ und $\mathrm{Var}(W_{ji}^{(L)}) = \frac{1}{m_{\text{out}}^{(L)}}$, was nach Mittelung genau der Xavier-Initialisierung von Satz 7.6 entspricht:

$$\mathrm{Var}(W_{ji}^{(L)}) = \frac{2}{m_{\text{in}}^{(L)} + m_{\text{out}}^{(L)}}$$

- Aktivierung mit der **logistischen Sigmoide** $h(a) = \sigma(a)$ kann durch (7.72) mit $c = d = 0.25$ und $u = 0.5$ approximiert werden und für $\mathrm{Var}(b_j^{(L)}) = 0$ erhält man für den Forward-Pass (7.80) entsprechend $\mathrm{Var}(W_{ji}^{(L)}) = \frac{16(\sigma_a^2 - \mathrm{Var}(b_j^{(L)}))}{m_{\text{in}}^{(L)} \cdot (\sigma_a^2+4)}$ und für den Backward-Pass (7.81) dagegen $\mathrm{Var}(W_{ji}^{(L)}) = \frac{16}{m_{\text{out}}^{(L)}}$. Da sich die Lösungen meist nicht sinnvoll mitteln lassen (σ_a^2 soll ja klein sein, damit die lineare Näherung für $\sigma(a)$ gut ist), verwendet man nur die letztere Lösung für den Backward-Pass (da die logistische Sigmoide meist nur in der Output-Layer verwendet wird, spielt der Forward-Pass eh keine Rolle) oder in Anlehnung an die Xavier-Initialisierung die "Mittelungs-Heuristik" [30]

$$\mathrm{Var}(W_{ji}^{(L)}) = \frac{32}{m_{\text{in}}^{(L)} + m_{\text{out}}^{(L)}}$$

- Aktivierung mit der relu**-artigen Funktionen** nach (7.72) für $u = 0$ und $\mathrm{Var}(b_j^{(L)}) = 0$ erhält man für Forward- und Backward Pass nach (7.80) und (7.81) entsprechend $\mathrm{Var}(W_{ji}^{(L)}) = \frac{2}{(c^2+d^2)m_{\text{in}}^{(L)}}$ und $\mathrm{Var}(W_{ji}^{(L)}) = \frac{2}{(c^2+d^2)m_{\text{out}}^{(L)}}$, oder nach Mittelung

$$\mathrm{Var}(W_{ji}^{(L)}) = \frac{4}{(c^2+d^2) \cdot (m_{\text{in}}^{(L)} + m_{\text{out}}^{(L)})} \tag{7.82}$$

  was für $c = 1$ und $d = 0$ der He-Initialisierung für $h = \text{relu}$ entspricht:

[30] Diese Mittelung hat offenbar keine theoretische Grundlage! Mit ihr folgen in der Notation der Sätze 7.6,7.8 die Varianz-Parameter und $\sigma = 4\sqrt{\frac{2}{m_{\text{in}}^{(L)}+m_{\text{out}}^{(L)}}}$ und $r = 4\sqrt{\frac{6}{m_{\text{in}}^{(L)}+m_{\text{out}}^{(L)}}}$ für $h(a) = \sigma(a)$.

Satz 7.8: He-Initialisierung

Um die Gefahr von Vanishing Gradients oder Exploding Gradients zu minimieren, initialisiert man sequentielle Neuronale Netzwerke mit relu Aktivierung, indem man alle Biase auf Null setzt, $b_j^{(L)} = 0$, und die synaptischen Gewichte jeder Verbindung $\mathbf{W}^{(L)}$ mit Zufallszahlen initialisert, sodass

$$E(W_{ji}^{(L)}) = 0 \qquad \text{und} \qquad \text{Var}(W_{ji}^{(L)}) = \frac{4}{m_{\text{in}}^{(L)} + m_{\text{out}}^{(L)}} \tag{7.83}$$

gilt, wobei $m_{\text{in}}^{(L)}$ und $m_{\text{out}}^{(L)}$ Fan-In und Fan-Out der Verbindung sind.

D.h. man kann die synaptischen Gewichte $W_{ji}^{(L)}$ z.B. mit gleichverteilten Zufallszahlen aus dem Intervall $[-r; r]$ oder mit Gauß-verteilten Zufallszahlen mit Erwartungswert $\mu = 0$ und Varianz σ^2 initialisieren wobei für die Verteilungs-Parameter gilt (siehe Anhang A.6,A.8):

$$r = \sqrt{\frac{12}{m_{\text{in}}^{(L)} + m_{\text{out}}^{(L)}}} \tag{7.84}$$

$$\sigma = \sqrt{\frac{4}{m_{\text{in}}^{(L)} + m_{\text{out}}^{(L)}}} \tag{7.85}$$

Beweis: (7.83) folgt direkt aus (7.82) für $c = 1$ und $d = 0$. (7.84) und (7.85) ergeben sich wie im Beweis von Satz 7.6. □

Bemerkungen:

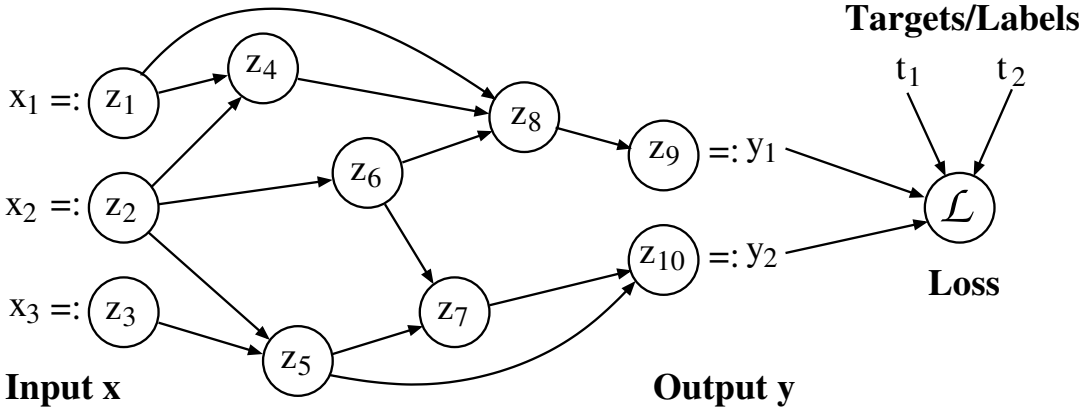
- Man nennt die He-Initialisierung oft auch Kaiming-Initialisierung, siehe Fußnote 28 auf Seite 198.
- Wie oben begründet verehindert He-Initialisierung Exploding und Vanishing Gradients nur in sequentiellen Netzwerken, falls die Fan-Ins und Fan-Outs sich nicht zu sehr unterscheiden. Trotzdem verwendet man in der Praxis häufig He-Initialisierung auch für nicht-sequentielle Netzwerke und/oder bei stark unterschiedlichen Fan-Ins und Fan-Outs.
- Ein Vorteil von relu Aktivierungen mit He-Initialisierung ist, dass die Theorie auch für große Varianzen der dendritischen Potentiale und Fehlerpotentiale korrekt ist, und keine Sättigungseffekte auftreten.
- Für Aktivierungen $h(a) = \sigma(a)$ mit der logistischen Sigmoidfunktion wird häufig $r = 4\sqrt{\frac{6}{m_{\text{in}}^{(L)}+m_{\text{out}}^{(L)}}}$ bzw. $\sigma = 4\sqrt{\frac{2}{m_{\text{in}}^{(L)}+m_{\text{out}}^{(L)}}}$ verwendet (siehe Fußnote 30 auf Seite 201).

Kapitel 8

Allgemeiner Backpropagation Algorithmus

8.1 Partielle Ableitungen von Funktionsgraphen

Der Backpropagation-Algorithmus für Neuronale Netze hat eine allgemeine Form, die unabhängig von den linearen Verbindungs-Modellen und speziellen Aktivierungsfunktionen der Neuronalen Netze ist. Allgemein betrachten wir eine Modell-Funktion $\mathbf{y}(\mathbf{x}; \mathbf{W})$ welche als ein azyklischer Graph aus beliebigen Funktions-Knoten z_j dargestellt werden kann. **Beispiel:** Bestimmen Sie zu folgendem Funktionsgraph die Modellfunktion $\mathbf{y}(\mathbf{x}; \mathbf{W})$.



Lösung: Jeder Funktions-Knoten z_j berechnet eine zugehörige Funktion $z_j(z_{i_1}, z_{i_2}, \ldots; \mathbf{w}_j)$, wobei $z_{i_1}, z_{i_2}, \ldots$ die Vorgänger-Knoten von z_j sind und $\mathbf{w}_j = \begin{pmatrix} w_{j1} & \ldots & w_{jR_j} \end{pmatrix}^{\mathrm{T}}$ der Modell-Parameter-Vektor von Funktion z_j ist, sodass $\mathbf{W} := (\mathbf{w}_1, \mathbf{w}_2, \ldots)$ als Liste solcher Parameter-Vektoren geschrieben werden

kann. Damit ist für Input $\mathbf{x} = \begin{pmatrix} x_1 & \dots & x_D \end{pmatrix}^{\mathrm{T}}$ die durch den Graph dargestellte Modellfunktion

$$\mathbf{y}(\mathbf{x}; \mathbf{W}) = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} z_9(z_8(x_1, z_4(x_1, x_2; \mathbf{w_4}), z_6(x_2; \mathbf{w}_6); \mathbf{w_8}); \mathbf{w_9}) \\ z_{10}(z_5(x_2, x_3; \mathbf{w}_5), z_7(z_5(x_2, x_3; \mathbf{w}_5), z_6(x_2; \mathbf{w}_6); \mathbf{w}_7)) \end{pmatrix}$$

wobei aus Gründen der Vereinheitlichung die Inputknoten $x_1, \dots, x_D$ mit den ersten Funktionsknoten $z_1, \dots, z_D$ indentifiziert werden (hier $D = 3$). Damit die Darstellung als Formel etwas übersichtlicher wird, kann man die Abhängigkeiten der Knoten z_j von den Modellparametern $\mathbf{w}_j$ auch weglassen (und sich dazudenken [1]):

$$\mathbf{y}(\mathbf{x}; \mathbf{W}) = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} z_9(z_8(x_1, z_4(x_1, x_2), z_6(x_2))) \\ z_{10}(z_5(x_2, x_3), z_7(z_5(x_2, x_3), z_6(x_2))) \end{pmatrix}$$

Die Verlust- oder Fehlerfunktion wird dann wie üblich in Abhängigkeit von den Outputs $\mathbf{y}$ und den Targets $\mathbf{t}$ berechnet:

$$\mathcal{L} := \mathcal{L}(\mathbf{y}, \mathbf{t})$$

Ziel ist es für eine gegebenes Trainingsdaten-Menge $\mathcal{D} := \{(\mathbf{x}_n, \mathbf{t}_n) | n = 1, \dots, N\}$ den Gesamtverlust des Modells

$$\mathcal{L}_{\mathcal{D}}(\mathbf{W}) := \sum_{n=1}^{N} \mathcal{L}(\mathbf{y}(\mathbf{x}_n; \mathbf{W}), \mathbf{t}_n)$$

zu minimieren, indem man die Modellparameter $\mathbf{W}$ optimiert. Um dies mittels stochastischem Gradientenabstieg zu tun (siehe ???) benötigt man den Teilgradienten $\nabla \mathcal{L}_n$ für den Teilfehler $\mathcal{L}_n := \mathcal{L}(\mathbf{y}(\mathbf{x}_n; \mathbf{W}), \mathbf{t}_n)$ der Modellprognose für einen Input $\mathbf{x}_n$. Der Gradientenvektor

$$\nabla \mathcal{L}_n = \begin{pmatrix} \cdots & \frac{\partial \mathcal{L}_n}{\partial w_{jr}} & \cdots \end{pmatrix}$$

enthält dabei alle partiellen Ableitungen von $\mathcal{L}_n$ bezüglich aller Modellparameter $w_{jr} := (\mathbf{w}_j)_r$. Deshalb ist das **Ziel des allgemeinen Backpropagation-Algorithmus** die effiziente Berechnung dieser partiellen Ableitungen.

Ein naives Berechnen mit Hilfe von Differenzen-Quotienten verbietet sich aus Effizienzgründen (siehe Argumentation in Kapitel ????). [2] Für ein effizientes Verfahren zur Berechnung von $\frac{\partial \mathcal{L}_n}{\partial w_{jr}}$ erinnern wir uns zunächst an folgende Sachverhalte:

[1] Man beachte, dass in obiger Skizze des Funktionsgraphs ebenfalls die Modellparameter $\mathbf{w}_j$ weggelassen wurden. Für einen vollständigen Graphen müsste man für jeden Funktionsknoten z_j einen zusätzlichen Vorgänger-Knoten $\mathbf{w}_j$ einzeichnen.

[2] Zur Erinnerung: Man müsste für jede partielle Ableitung $\frac{\partial \mathcal{L}_n}{\partial w_{jr}} = \lim_{\epsilon \to 0} \frac{\mathcal{L}_n(w_{jr} + \epsilon) - \mathcal{L}_n(w_{jr})}{\epsilon}$ die Modellfunktion $\mathbf{y}(\mathbf{x}_n)$ mit um ϵ verändertem Modellparameter w_{jr} testen, sodass der Rechenaufwand (mindestenes) quadratisch $O(N_P^2)$ in der Anzahl der Modellparameter N_P skalieren würde (man bräuchte N_P solche Tests und für jeden Test fließt jeder der N_P Modellparameter mindestens einmal in die Berechnung ein). Dagegen werden wir sehen, dass der Backpropagation-Algorithmen mit $O(N_P)$ linear in N_P bleibt.

- Für partielle Ableitungen $\frac{\partial \mathcal{L}_n}{\partial w_{jr}}$ darf man so tun, als existiere nur die Variable w_{jr} nach der abgeleitet wird, während man alle anderen Variablen als Konstanten auffassen darf. D.h. wir betrachten hierfür die Funktion $\mathcal{L}_n(w_{jr})$.
- Da $\mathcal{L}_n$ nicht direkt von w_{jr} abhängt, sondern dem Netzwerk-Graph entsprechend über die nachfolgenden Größen z_j können wir $\mathcal{L}_n(w_{jr})$ auch als geschachtelte Funktion

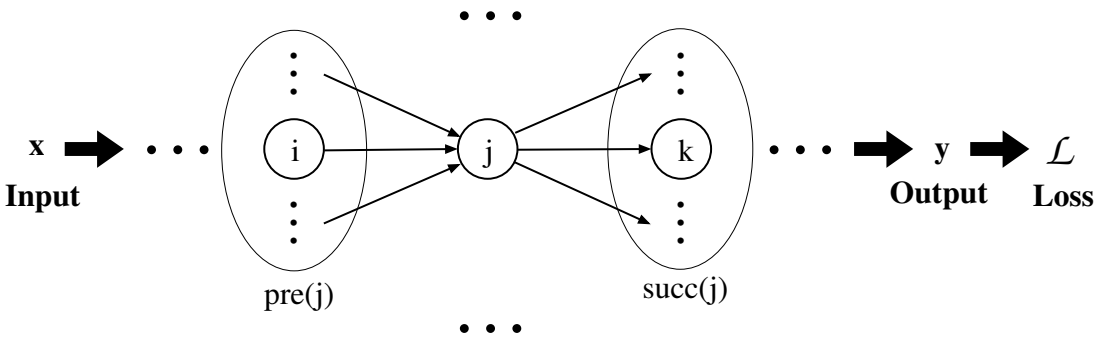
$$\mathcal{L}_n = \mathcal{L}_n(z_j(w_{jr})) \tag{8.1}$$

schreiben, da $\mathcal{L}_n$ unabhängig von w_{jr} wird, sobald z_j gegeben ist. [3]

- Mit der gleichen Argumentation kann man $\mathcal{L}_n$ auch entsprechend der folgenden Skizze der Abhängigkeiten als Funktion der Nachfolgeknoten z_k von z_j mit $k \in \text{succ}(j)$ schreiben:

$$\mathcal{L}_n = \mathcal{L}_n \left(\bigoplus_{k \in \text{succ}(j)} z_k(z_j(w_{jr})) \right) \tag{8.2}$$

Hierbei ist succ(j) die Indexmenge der Nachfolgeknoten von z_j, z.B. in dem Funktionsgraph der vorigen Beispielaufgabe gilt succ$(2) = \{4, 5, 6\}$. Ferner ist $\oplus$ die Vektor/Variablen-Konkatenation, z.B. $\oplus_{k \in \{4,5,6\}} z_k = \begin{pmatrix} z_4 & z_5 & z_6 \end{pmatrix}^{\mathrm{T}}$.



Damit können wir den Allgemeinen Backpropagation-Algorithmus für beliebige Funktionsgraphen herleiten: Aus der Darstellung (8.1) folgt durch partielles Ableiten mit der 1D-Kettenregel (siehe ????)

$$\frac{\partial \mathcal{L}_n}{\partial w_{jr}} = \frac{\partial \mathcal{L}_n}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_{jr}} \quad = \delta_j \cdot \zeta_{jr} \qquad \text{für} \tag{8.3}$$

$$\delta_j := \frac{\partial \mathcal{L}_n}{\partial z_j} \qquad \text{und} \qquad \zeta_{jr} := \frac{\partial z_j}{\partial w_{jr}} \ , \tag{8.4}$$

[3] Für den Sachverhalt $\mathcal{L}_n$ unabhängig von w_{jr} gegeben z_j schreibt man kurz $\mathcal{L}_n \perp\!\!\!\perp w_{jr} \mid z_j$. Allgemein gilt in einem Funktionsgraphen $f \perp\!\!\!\perp g \mid h$ genau dann, wenn alle Pfade von g nach f durch h verlaufen, und somit g irrelevant wird, sobald h gegeben ist. Wir werden dieses Konzept Bedingte Unabhängigkeit später in Kapitel ??? über Graphische Probabilistische Modelle in allgemeinerer Form behandeln.

d.h. die gesuchte partielle Ableitung $\frac{\partial \mathcal{L}_n}{\partial w_{jr}}$ läßt sich ganz allgemein als Produkt zweier partieller Ableitungen δ_j und ζ_{jr} schreiben. [4] Hier folgt der zweite Faktor ζ_{jr} aus der Form der Funktion z_j. Den ersten Faktor δ_j nennt man (verallgemeinertes) Fehlersignal. Wie bei (5.15) in Kap. 5 muss es für die Output-Knoten y_k aus der Form der Fehlerfunktion $\mathcal{L}_n$ initialisiert werden, während es für innere Knoten z_j rekursiv berechnet werden kann: Denn aus der Darstellung (8.2) folgt durch partielles Ableiten von $\mathcal{L}_n(\oplus_{k\in \mathrm{succ}(j)} z_k(z_j))$ nach z_j mit der mehrdimensionalen Kettenregel (siehe ???)

$$\begin{aligned}\delta_j := \frac{\partial \mathcal{L}_n}{\partial z_j} &= \frac{\partial \mathcal{L}_n}{\partial \oplus_{k\in \mathrm{succ}(j)}\, z_k} \cdot \frac{\partial \oplus_{k\in \mathrm{succ}(j)}\, z_k}{\partial z_j} \\ &= \sum_{k\in \mathrm{succ}(j)} \frac{\partial \mathcal{L}_n}{\partial z_k} \cdot \frac{\partial z_k}{\partial z_j} = \sum_{k\in \mathrm{succ}(j)} \delta_k \cdot \frac{\partial z_k}{\partial z_j} \end{aligned} \tag{8.5}$$

Man beachte dass (8.5) wieder eine rekursive Gleichung ist, mit der man die Fehlersignale von der Output-Layer bis zur Input-Layer rückpropagieren kann. Zusammen mit der Vorwärtspropagation der Funktionswerte z_j haben wir damit also den folgenden allgemeinen Backpropagation-Algorithmus hergeleitet:

Satz 8.1: Allgemeiner Error-Backpropagation-Algorithmus für Modellparameter-Gradienten

Gegeben sei eine Modellfunktion $\mathbf{y}(\mathbf{x};\mathbf{W})$ mit Modellparametern $\mathbf{W} = (\mathbf{w}_1, \mathbf{w}_2, \ldots)$ durch einen zyklenfreier Funktionsgraphen mit Funktionsknoten z_j (wie in der Skizze auf Seite **??**). Für Trainings-Daten $\{(\mathbf{x}_n, \mathbf{t}_n)|n = 1, \ldots, N\}$ mit Verlust-Funktion $\mathcal{L}(\mathbf{W}) = \sum_{n=1}^{N} \mathcal{L}_n(\mathbf{y}(\mathbf{x}_n;\mathbf{W}), \mathbf{t}_n)$ kann man dann den Teil-Gradienten $\nabla\mathcal{L}_n(\mathbf{W})$ zum n-ten Inputvektor $\mathbf{x}_n$ durch den **Allgemeinen Backpropagation-Algorithmus** berechnen:

I) **Vorwärtspropagation der Funktionswerte für Input $\mathbf{x}_n$:**

a) Setze die Aktivität $z_i := x_{n,i}$ von allen Input-Knoten i entsprechend dem Datenvektor $\mathbf{x}_n$.

b) Berechne ausgehend von der Input-Layer bis zur Output-Layer alle weiteren Funktions-Werte z_j aller restlichen Funktions-Knoten j mittels

$$z_j = z_j\left(\bigoplus_{i\in \mathrm{pre}(j)} z_i; \mathbf{w}_j\right) . \tag{8.6}$$

[4] Für den Spezialfall von Neuronalen Netzen aus Kapitel 5 entspricht δ_j der Definition des Fehlersignals $\delta_j := \frac{\partial E_n}{\partial a_j}$ von Seite 122 falls $z_j := a_j := \sum_r w_{jr} z_r$ ein Aktivitätsknoten ist, der ein dendritisches Potential berechnet, und $\zeta_{jr} = z_r$ entspricht dann der präsynaptischen Feuerrate. D.h. es folgt (5.9) als Spezialfall aus (8.3).

II) **Rückwärtspropagation (=Backpropagation) der Fehler-Terme** δ_j:

a) Initialisiere die Fehler-Terme der Output-Knoten k nach (8.4) mittels

$$\delta_k := \frac{\partial \mathcal{L}_n}{\partial z_k} \ . \tag{8.7}$$

b) Berechne ausgehend von der Output-Layer bis zur Input-Layer nach (8.5) die restlichen Fehlerterme der inneren Funktionsknoten j mit der Rekursion

$$\delta_j := \sum_{k \in \mathrm{succ}(j)} \frac{\partial z_k}{\partial z_j} \cdot \delta_k \ . \tag{8.8}$$

III) **Berechnung des Gradienten der Fehlerfunktion**: Der Gradient ergibt sich dann nach (8.3) aus den partiellen Ableitungen

$$\frac{\partial \mathcal{L}_n}{\partial w_{jr}} = \delta_j \cdot \zeta_{jr} \qquad \text{für} \quad \zeta_{jr} := \frac{\partial z_j}{\partial w_{jr}} \ . \tag{8.9}$$

Damit kann man den **Parameter-Update durch Gradientenabstieg** mit Lernrate $\eta > 0$ zur Zeit τ ähnlich wie in Def. 3.8 auf Seite 56 bestimmen:

$$w_{jr}^{(\tau+1)} = w_{jr}^{(\tau)} - \eta \frac{\partial \mathcal{L}_n}{\partial w_{jr}} \tag{8.10}$$

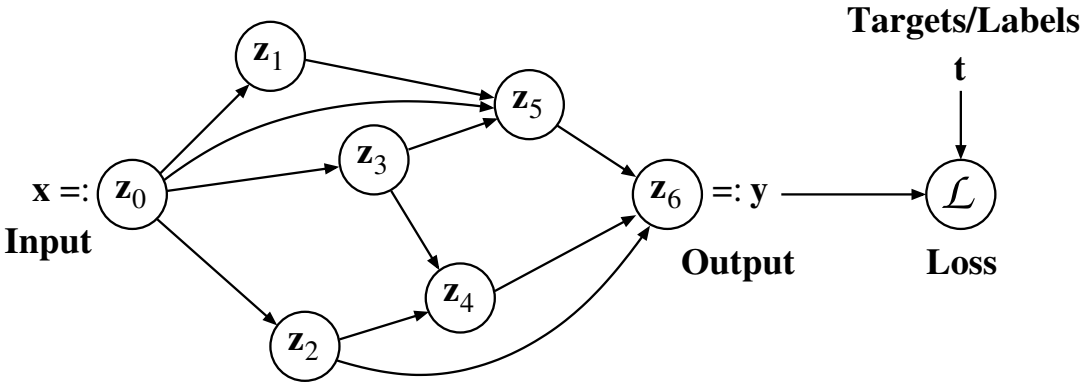
Weitere Aufgaben für Übungen: 1) Bestimmen Sie die Modellfunktion für ein oder zwei weitere Beispiele 2) Zeigen Sie: Man kann jeden azyklischen Funktionsgraph als Sequenz von Layern darstellen. 3) Berechnen Sie die den Gradientenvektor für eine geschachtelte Funktion: a) Zeichnen Sie den Funktionsgraph. b) Wenden Sie Backprop an c) Vergleichen Sie mit dem Differnenquotienten d) Versuchen Sie von Hand Ableitung auszurechnen und vergleichen Sie!

8.2 Funktionsgraphen mit Layer-Knoten: Vektor-/Matrix-Darstellung

In Kapitel 8.1 waren die Knoten der Funktionsgraphen z_j "eindimensionale" Funktionen vom Typ $z_j : \mathbb{R}^{N_j} \to \mathbb{R}$, die zwar von beliebig vielen Input Variablen $N_j \in \mathbb{N}$ abhängen durften, aber nur eine Output-Komponente $z_j \in \mathbb{R}$ besaßen. Der Vorteil dieser Darstellung ist, dass man bei der Herleitung des Backpropagation-Algorithmus fast ohne die mehrdimensionale Differentialrechnung auskommt, und deshalb der Stoff vielen Studierenden besser zugänglich ist (einzige Ausnahme war Gleichung (8.5) für die man die mehrdimensionale Kettenregel (**??**) benötigt). In der Praxis hat man es allerdings meist mit geschichteten Funktionsgraphen zu tun, bei denen die Knoten-Funktionen

vom Typ $\mathbf{z}_j : \mathbb{R}^{N_j} \to \mathbb{R}^{M_j}$ sind mit $N_j, M_j \in \mathbb{N}$. D.h. jeder Knoten entspricht einer ganzen Schicht (engl. Layer) aus beliebig vielen Output-Komponenten, und wird deshalb als (fettgedruckter) Spalten-Vektor $\mathbf{z}_j$ dargestellt. Dieser Fall kann zwar auch wie in Kapitel 8.1 dargestellt werden, indem man jeden "Vektor-Knoten" $\mathbf{z}_j$ durch M_j "einfache" Knoten z_{j_i} ersetzt, aber dadurch wird der Graph unnötig groß und unübersichtlich. Deshalb wollen wir hier den Backpropagation-Algorithmus nochmal in der viel übersichtlicheren Vektor-Darstellung herleiten. Dazu sollten Sie unbedingt die Grundlagen zur mehrdimensionalen Differentialrechnung verstanden haben. [5]

Ein Funktionsgraph in Vektor/Layer-Darstellung sieht dann z.B. so aus wie in der folgenden Skizze. Im Unterschied zur Skizze auf Seite 203 sind die Knoten fettgedruckt, und wir fassen alle Input-Knoten in einer Layer $\mathbf{x} =: \mathbf{z}_0$ sowie alle Output-Knoten in einer Layer $\mathbf{y} := \mathbf{z}_L$ zusammen, wobei $L \in \mathbb{N}$ die Anzahl der Layer (ohne die Input-Layer) ist (hier: $L = 6$). Die inneren Knoten $\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_{\mathbf{L-1}}$ bezeichnen wir hierbei auch als Hidden Layer (deutsch "versteckte Schicht), da sie keine direkte Verbindung zu den Input-/Ouput Knoten $\mathbf{x}$ und $\mathbf{y}$ haben, und deshalb von außen nicht sichtbar sind.



Der in der Skizze dargestellte Funktionsgraph entspricht dann beispielsweise der geschachtelten Funktion

$$\mathbf{y}(\mathbf{x}; \mathbf{W}) = \mathbf{z}_6(\mathbf{z}_2(\mathbf{x}), \mathbf{z}_4(\mathbf{z}_2(\mathbf{x}), \mathbf{z}_3(\mathbf{x})), \mathbf{z}_5(\mathbf{x}, \mathbf{z}_1(\mathbf{x}), \mathbf{z}_3(\mathbf{x}))) \ .$$

Wie in Kap. **??** ist es das Ziel wieder den Gesamtverlust

$$\mathcal{L}_{\mathcal{D}}(\mathbf{W}) := \sum_{n=1}^{N} \mathcal{L}(\mathbf{y}(\mathbf{x}_n; \mathbf{W}), \mathbf{t}_n)$$

[5] Erinnern Sie sich dazu bitte, dass $\frac{\partial \mathbf{a}}{\partial \mathbf{b}}$ die Ableitung der Vektor-Funktion $\mathbf{a} \in \mathbb{R}^M$ nach dem Variablen-Vektor $\mathbf{b} \in \mathbb{R}^N$, und damit eine **Jacobi-Matrix** der Größe $M \times N$ ist (siehe (**??**)). Die Komponente in Zeile i und Spalte j ist dabei die (eindimensionale) parteille Ableitung $\left(\frac{\partial \mathbf{a}}{\partial \mathbf{b}}\right)_{ij} = \frac{\partial a_i}{\partial b_j}$ der i-ten Funktionskomponente a_i nach der j-ten Variable b_j. Mit dieser Notation gelten in weiterhin alle bekannten Ableitungsregeln, wie Linearität (**??**) und Kettenregel (**??**), wobei man statt der 1D-Multiplikation die Matrizenmultiplikation benutzen muss.

des Modells zu minimieren, wozu man den Gradientenvektor

$$\nabla \mathcal{L}_n(\mathbf{W}) = \frac{\partial \mathcal{L}_n}{\oplus_{j=1}^{L} \mathbf{w}_j} = \left(\frac{\partial \mathcal{L}_n}{\partial \mathbf{w}_1} \quad \cdots \quad \frac{\partial \mathcal{L}_n}{\partial \mathbf{w}_L} \right) = \left(\cdots \quad \frac{\partial \mathcal{L}_n}{\partial w_{jr}} \quad \cdots \right)$$

aller partiellen Ableitungen $\frac{\partial \mathcal{L}_n}{\partial w_{jr}}$ bestimmen muss. In Analogie zu (8.1) und (8.2) kann man hierfür $\mathcal{L}_n(\mathbf{w}_j)$ wieder als geschachtelte Funktion (siehe Skizze auf Seite 205)

$$\mathcal{L}_n = \mathcal{L}_n(\mathbf{z}_j(\mathbf{w}_j)) \qquad \text{oder} \tag{8.11}$$

$$\mathcal{L}_n = \mathcal{L}_n \left(\bigoplus_{k \in \text{succ}(j)} \mathbf{z}_k(\mathbf{z}_j(\mathbf{w}_j)) \right) \tag{8.12}$$

auffassen. Aus der ersten Darstellung (8.11) folgt durch (mehrdimensionales) partielles Ableiten mit der mehrdimensionalen Kettenregel (**??**)

$$\frac{\partial \mathcal{L}_n}{\partial \mathbf{w}_j} = \frac{\partial \mathcal{L}_n}{\partial \mathbf{z}_j} \cdot \frac{\partial \mathbf{z}_j}{\partial \mathbf{w}_j} \quad = \boldsymbol{\delta}_j^{\mathrm{T}} \cdot \boldsymbol{\zeta}_j \qquad \text{für} \tag{8.13}$$

$$\boldsymbol{\delta}_j := \left(\frac{\partial \mathcal{L}_n}{\partial \mathbf{z}_j} \right)^{\mathrm{T}} \qquad \text{und} \qquad \boldsymbol{\zeta}_j := \frac{\partial \mathbf{z}_j}{\partial \mathbf{w}_j} \,, \tag{8.14}$$

wobei hier das Fehlersignal $\boldsymbol{\delta}_j$ ein zum Spaltenvektor transponierter Gradientenvektor der Länge M_j ist, und $\boldsymbol{\zeta}_j$ eine $M_j \times R_j$ Jacobi-Matrix ist. D.h. ähnlich wie in (8.3) kann man die partiellen Ableitungen als (Matrix-)Produkt zweier partieller Ableitungen δ_j und ζ_{jr} schreiben. [6] Auch die Backpropagation-Rekursion ergibt sich für Vektor-Knoten in Analogie zu (8.5): Aus der Darstellung (8.12) folgt mit der mehrdimensionalen Kettenregel (**??**)

$$\begin{aligned} \boldsymbol{\delta}_j^{\mathrm{T}} := \frac{\partial \mathcal{L}_n}{\partial \mathbf{z}_j} &= \frac{\partial \mathcal{L}_n}{\partial \oplus_{k \in \text{succ}(j)} \mathbf{z}_k} \cdot \frac{\partial \oplus_{k \in \text{succ}(j)} \mathbf{z}_k}{\partial \mathbf{z}_j} \\ &= \sum_{k \in \text{succ}(j)} \frac{\partial \mathcal{L}_n}{\partial \mathbf{z}_k} \cdot \frac{\partial \mathbf{z}_k}{\partial \mathbf{z}_j} = \sum_{k \in \text{succ}(j)} \boldsymbol{\delta}_k^{\mathrm{T}} \cdot \frac{\partial \mathbf{z}_k}{\partial \mathbf{z}_j} \end{aligned} \tag{8.15}$$

Damit ist der folgende Satz gezeigt:

Satz 8.2: Allgemeiner Error-Backpropagation-Algorithmus für Funktionsgraphen mit Vektor/Layer-Knoten

Gegeben sei eine Modellfunktion $\mathbf{y}(\mathbf{x}; \mathbf{W})$ mit Modellparametern $\mathbf{W} = (\mathbf{w}_1, \mathbf{w}_2, \ldots)$ durch einen zyklenfreier Funktionsgraphen mit Vektor-

[6] Man beachte, dass damit die einzelnen partiellen Ableitungen $\frac{\partial \mathcal{L}_n}{\partial w_{jr}} = \left(\boldsymbol{\delta}_j^{\mathrm{T}} \cdot \boldsymbol{\zeta}_j \right)_r = \sum_{i=1}^{M_j} (\boldsymbol{\delta}_j)_i \cdot (\boldsymbol{\zeta}_j)_{ir} = \sum_{i=1}^{M_j} \frac{\partial \mathcal{L}_n}{\partial (\mathbf{z}_j)_i} \cdot \frac{\partial (\mathbf{z}_j)_i}{\partial w_{jr}}$ sind. D.h. weile alle Komponenten $i = 1, ..., M_j$ des Vektor-Knoten $\mathbf{z}_j$ von denselben Modell-Parametern $\mathbf{w}_j$ abhängen, summieren die partiellen Ableitungen im Gegensatz zu (8.3) über alle diese Komponenten von $\mathbf{z}_j$.

Funktionsknoten $\mathbf{z}_j$ (wie in der Skizze auf Seite **??**). Für Trainings-Daten $\{(\mathbf{x}_n, \mathbf{t}_n) | n = 1, \ldots, N\}$ mit Verlust-Funktion $\mathcal{L}(\mathbf{W}) = \sum_{n=1}^{N} \mathcal{L}_n(\mathbf{y}(\mathbf{x}_n; \mathbf{W}), \mathbf{t}_n)$ kann man dann den Teil-Gradienten $\nabla \mathcal{L}_n(\mathbf{W})$ zum n-ten Inputvektor $\mathbf{x}_n$ durch den **Allgemeinen Backpropagation-Algorithmus für Vektor-Knoten** berechnen:

I) **Vorwärtspropagation der Funktionswerte für Input $\mathbf{x}_n$**:

a) Setze die Aktivität $\mathbf{z}_0 := \mathbf{x}_n$ der Input-Layer entsprechend dem Datenvektor $\mathbf{x}_n$.

b) Berechne ausgehend von der Input-Layer bis zur Output-Layer alle weiteren Funktions-Werte z_j aller restlichen Funktions-Knoten j mittels

$$\mathbf{z}_j = \mathbf{z}_j \left(\bigoplus_{i \in \mathrm{pre}(j)} \mathbf{z}_i; \mathbf{w}_j \right) . \tag{8.16}$$

II) **Rückwärtspropagation (=Backpropagation) der Fehler-Terme $\boldsymbol{\delta}_j$**:

a) Initialisiere die Fehler-Terme der Output-Layer L nach (8.14) mittels

$$\boldsymbol{\delta}_L^{\mathrm{T}} := \frac{\partial \mathcal{L}_n}{\partial \mathbf{z}_L} . \tag{8.17}$$

b) Berechne ausgehend von der Output-Layer bis zur Input-Layer nach (8.15) die restlichen Fehlerterme der inneren Funktionsknoten j mit der Rekursion

$$\boldsymbol{\delta}_j^{\mathrm{T}} := \sum_{k \in \mathrm{succ}(j)} \boldsymbol{\delta}_k^{\mathrm{T}} \frac{\partial \mathbf{z}_k}{\partial \mathbf{z}_j} . \tag{8.18}$$

III) **Berechnung des Gradienten der Fehlerfunktion**: Der Gradient ergibt sich dann nach (8.13) aus den partiellen Ableitungen

$$\mathbf{g}^{\mathrm{T}} := \frac{\partial \mathcal{L}_n}{\partial \mathbf{w}_j} = \boldsymbol{\delta}_j^{\mathrm{T}} \cdot \boldsymbol{\zeta}_j \qquad \text{für} \quad \boldsymbol{\zeta}_j := \frac{\partial \mathbf{z}_j}{\partial \mathbf{w}_j} . \tag{8.19}$$

Damit kann man den **Parameter-Update durch Gradientenabstieg** mit Lernrate $\eta > 0$ zur Zeit τ ähnlich wie in Def. 3.8 auf Seite 56 bestimmen:

$$\mathbf{w}_j^{(\tau+1)} = \mathbf{w}_j^{(\tau)} - \eta \mathbf{g} \tag{8.20}$$

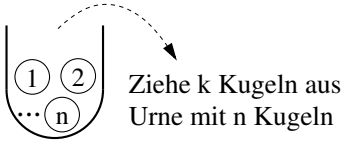
Anhang A

Kombinatorik und Wahrscheinlichkeitsrechnung

A.1 Elementare Kombinatorik

Um die Wahrscheinlichkeit eines Ereignisses A zu bestimmen zählt man oft wieviele mögliche Ausgänge N eines Vorganges es geben kann. Unter der Annahme, dass alle N Ausgänge gleich wahrscheinlich sind ist die Wahrscheinlichkeit von A einfach der Kehrwert $1/N$. **Beispiel:** Wieviele Möglichkeiten gibt im Lotto 6 der 49 Kugeln einer Urne zu ziehen? Antwort: Siehe folgende Skizze (a) und Tabelle (III)...

(a) Urnenmodell



(b) Zellenbelegungsmodell

Lege k Marken (=Kugeln) in die n Zellen

d.h. ordne k+n−1 Dinge (Marken, Trennwaende) in einer Reihe an

Die Anzahl der Möglichkeiten k Kugeln aus einer Urne mit n Kugeln zu ziehen (siehe Skizze (a)) wird in folgender Tabelle dargestellt (vgl. Skript Mathe-2, Satz ???):

	ohne Zurücklegen	mit Zurücklegen
mit Beachten der Reihenfolge	(I): $n(n-1)\dots(n-k+1) = \frac{n!}{(n-k)!}$	(II): n^k
ohne Beachten der Reihenfolge	(III): $\frac{n!}{k!(n-k)!} =: \binom{n}{k}$	(IV): $\binom{n+k-1}{k}$

Beweis: (I) Falls auf die Reihenfolge der Ziehung geachtet wird und die Kugeln nicht in die Urne zurückgelegt werden gilt folgendes: Für das Ziehen der 1.Kugel hat man n Möglichkeiten, für die 2.Kugel noch $n-1$ Möglichkeiten, ..., und für das Ziehen

der k.Kugel noch $n-k+1$ Möglichkeiten. Die sich die verschiedenen Möglichkeiten unabhängig kombinieren lassen, gibt es insgesamt also $n(n-1)\dots(n-k+1)$ viele Möglichkeiten. **(II)** Der Fall, dass auf die Reihenfolge der Ziehung geachtet wird und die Kugeln nach jedem Ziehen zurückgelegt werden ist besonders einfach: Für das Ziehen der 1.Kugel gibt es n Möglichenkeiten, für die 2.Kugel ebenso n Möglichkeiten, ..., für die k.Kugel ebenso n Möglichkeit, insgesamt also $n \cdot n \cdot \ldots \cdot n = n^k$ viele Möglichkeiten. **(III)** Für Ziehen ohne Zurücklegen und ohne Beachten der Reihenfolge gilt: Der Vorgang ist ähnlich wie in (I) aber, da nicht auf die Anordnung geachtet, zählen alle Ergebnisse mit denselben Kugeln als identisch. Da man k Kugeln auf $k(k-1)\cdots 1 = k!$ Weisen anordnen kann, werden jeweils $k!$ Möglichkeiten nach (I) zu einer Möglichkeite nach (III) zusammengefasst. Deshalb gibt es Faktor $1/k!$ weniger Möglichkeiten als bei (I), also $\frac{n!}{(n-k)!}\frac{1}{k!}$. **(IV)** Das Ziehen mit Zurücklegen ohne Reihenfolge versteht man am einfachsten mit Hilfe des sogenannten Zellenbelegungsmodells (obige Skizze (b)): Hierbei hat man n Zellen, die durch $n-1$ Trennwände abgetrennt sind. Man muss nun k Marken (entsprechen den Kugeln im Urnenmodell) auf die Zellen verteilen, wobei eine Zelle mehrere Marken enthalten darf. Hierfür gibt es genauso viele Möglichkeiten wie man die $n-1$ Trennwände und k Marken von links nach rechts in eine Zeile zeichnen kann. Für eine Möglichkeit muss man also aus den $n-1+k$ Dingen (=freie Plätze in der Zeile für Trennwände und Marken) k Dinge für die Marken auswählen. Nach (III) gibt es hierfür genau $\binom{n-1+k}{k}$ Möglichkeiten. □

Beispiel: Um im Lotto 6 aus 49 Kugeln auszuwählen (ohne Reihenfolge, ohne Zurücklegen) gibt es also genau $N = \binom{49}{6} = \frac{49\cdot 48\cdots 44}{6\cdot 5\cdots 1} = 13983816$ Möglichkeiten. Da alle Möglichkeiten gleich wahrscheinlich sind ist die Wahrscheinlichkeit für das Ereignis $A :=$"*6 richtige im Lotto*" also $P(A) = \frac{1}{N} = \frac{1}{13983816} \approx 7.15 \cdot 10^{-8}$.

Im folgenden Verallgemeinern wir diese Idee von Wahrscheinlichkeiten auf allgemeine Wahrscheinlichkeitsräume, in denen nicht jedes Ereignis dieselbe Wahrscheinlichkeit $1/N$ haben muss.

A.2 Wahrscheinlichkeitsraum

Ein Wahrscheinlichkeitsraum (Ω, Σ, P) besteht aus Ergebnismenge Ω, Ereignissystem Σ und Wahrscheinlichkeitsmaß P:

- Ergebnismenge Ω, welche alle möglichen elementaren Ausgänge ω eines Zufallsexperiments enthält. Ein $\omega \in \Omega$ nennt man auch Elementarereignis.
- Ereignissystem Σ, welche "sinnvolle" Teilmengen von Ω enthält. Eine Teilmenge $A \subseteq \Omega$ nennt man auch Ereignis. Σ bildet eine sogenannte σ-Algebra und muss abgeschlossen bzgl. der Bildung von Komplementen und (abzählbaren) Vereinigungen sein. D.h. falls $A \in \Sigma$, dann muss auch $A^c := \Omega - A \in \Sigma$ sein; falls $A_1, A_2, \ldots \in \Sigma$, dann muss auch $A_1 \cup A_2 \cup \ldots \in \Sigma$ sein.

- Das Wahrscheinlichkeitsmaß $P : \Sigma \to [0;1]$ ordnet jedem Ereignis $A \subseteq \Omega$ eine Wahrscheinlichkeit $P(A)$ zu. P muß stets die drei Kolmogoroff-Axiome erfüllen:

  I) Es gilt $P(A) \in [0;1]$ für jedes Ereignis $A \in \Sigma$.

  II) Es gilt $P(\varnothing) = 0$ und $P(\Omega) = 1$.

  III) Falls A und B disjunkte Ereignisse mit $A \cap B = \varnothing$ sind, dann gilt $P(A \cup B) = P(A) + P(B)$.

  Aus den Kolmogoroff-Axiomen folgt für beliebige $A, B \in \Sigma$ unmittelbar $P(A \cup B) = P(A) + P(B) - P(A \cap B)$. [1]

- Ein Tupel (Ω, Σ) aus Ergebnismenge Ω und Ereignissystem Σ (aber ohne Angabe eines Wahrscheinlichkeitsmaß) nennt man einen Messraum.

Beispiel: Beim Werfen eines Würfels sind die Elementarereignisse $\Omega = \{1, 2, 3, 4, 5, 6\}$, die Ereignismenge ist $\Sigma = \mathcal{P}(\Omega)$ und $P(\{\omega\}) = \frac{1}{6}$ für alle $\omega \in \Omega$. Das Ereignis $A := \{2, 4, 6\}$ dass eine gerade Zahl gewürfelt wird ist $P(A) = P(2)+P(4)+P(6) = 3/6 = 0.5$.

A.3 Zufallsvariablen

Der Begriff einer Zufallsvariable wird wie folgt definiert:

- Sei (Ω, Σ, P) ein Wahrscheinlichkeitsraum und (Ω', Σ') ein Messraum. Dann nennt man eine Abbildung $X : \Omega \to \Omega'$ eine Ω'-Zufallsvariable auf Ω.
- Falls klar ist welche Ergebnismengen jeweils gemeint sind, dann nennt man X einfach eine Zufallsvariable. Statt dem Funktionswert $X(\omega)$ zu einem zufälligen Elementarereignis $\omega \in \Omega$ schreibt man oft einfach X.
- Die Wahrscheinlichkeit des Ereignis $A' \in \Sigma'$ ist dann durch $P(\{\omega | X(\omega) \in A'\})$ bestimmt. Statt $P(\{\omega | X(\omega) \in A'\})$ schreibt man oft einfach $P(X \in A')$.

Beispiele: Beim Werfen zweier Würfel ist die Ergebnismenge $\Omega = \{1, 2, 3, 4, 5, 6\}^2 = \{(n_1, n_2) | n_1, n_2 = \{1, 2, 3, 4, 5, 6\}\}$, die Ereignismenge $\Sigma = \mathcal{P}(\Omega)$ und das Wahrscheinlichkeitsmaß $P(\{\omega\}) = 1/36$ für alle $\omega \in \Omega$. Wir können verschiedene Zufallsvariablen definieren:
a) Die Zufallsvariable $X((n_1, n_2)) := n_1 + n_2$ summiert die Augenzahl der beiden Würfel. D.h. $\Omega' = \{2, 3, \ldots, 12\}$ und $\Sigma' = \mathcal{P}(\Omega')$. Die Wahrscheinlichkeit, dass man eine 11 oder eine 12 würfelt ist dann $P(X \in \{11, 12\}) = P(\{(5, 6), (6, 5), (6, 6)\}) = 3/36 = 1/12$. Die

[1] Da B als disjunkte Vereinigung $(A \cap B) \cup (B \backslash A)$ dargestellt werden kann, gilt $P(B) = P(A \cap B) + P(B \backslash A)$ nach Axiom (III). Umformen ergibt $P(B \backslash A) = P(B) - P(A \cap B)$. Da $A \cup B$ als disjunkte Vereinigung $A \cup (B \backslash A)$ dargestellt werden kann folgt mit Axiom (III) wie gewünscht $P(A \cup B) = P(A) + P(B \backslash A) = P(A) + P(B) - P(A \cap B)$. □

Wahrscheinlichkeit einer 3 ist $P(X = 3) = P(\{(1, 2), (2, 1)\}) = 2/36 = 1/18$.
b) Die Zufallsvariable $Y((n_1, n_2)) := n_1$ liefert die Augenzahl des ersten Würfels. Die Wahrscheinlichkeit einer 1 ist $P(Y = 1) = P(\{(1, n_2)|n_2 \in \{1, 2, 3, 4, 5, 6\}\}) = 6/36 = 1/6$.
c) Die Zufallsvariable $Z((n_1, n_2)) := n_2$ liefert die Augenzahl des zweiten Würfels. Die Wahrscheinlichkeit einer 6 ist $P(Z = 1) = P(\{(n_1, 6)|n_1 \in \{1, 2, 3, 4, 5, 6\}\}) = 6/36 = 1/6$.

A.4 Verteilung, Erwartungswert und Varianz von diskreten Zufallsvariablen

Falls der Wertebereich Ω' der Zufallsvariablen höchstens abzählbar viele Elemente enthält, dann spricht man von diskreten Zufallsvariablen. Als Verteilung einer diskreten Zufallsvariablen X bezeichnet man die Abbildung

$$p_X : \Omega' \to [0; 1] \tag{A.1}$$

welche jedem möglichen Wert $x \in \Omega'$ von X die entsprechende Wahrscheinlichkeit $p_X(x) := P(X = x)$ zuweist.
Beispiele: Falls eine Zufallsvariable nur endlich viele Werte annimmt ($|\Omega'| < \infty$), dann kann man ihre Verteilung als Tabelle darstellen. Z.B. gilt für die Verteilung $p(x)$ der Summe X zweier Würfel (siehe obiges Beispiel a): [2]

x	2	3	4	5	6	7	8	9	10	11	12
$p(x)$	$\frac{1}{36}$	$\frac{2}{36}$	$\frac{3}{36}$	$\frac{4}{36}$	$\frac{5}{36}$	$\frac{6}{36}$	$\frac{5}{36}$	$\frac{4}{36}$	$\frac{3}{36}$	$\frac{2}{36}$	$\frac{1}{36}$

Man interessiert sich oft für das "durchschnittliche" Ergebnis einer Zufallsvariablen. Würfelt man z.B. n mal mit zwei Würfeln, so erhält man ungefähr $np(2)$ mal den Wert $X = 2$, $np(3)$ mal den Wert $X = 3$, ..., und $np(12)$ mal den Wert 12. Insgesamt summiert sich die Augenzahl also auf ungefähr $np(2) \cdot 2 + np(3) \cdot 3 + \ldots + np(12) \cdot 12$. Den Durchschnittswert pro Wurf erhält man nach Teilen durch n als $p(2) \cdot 2 + \ldots + p(12) \cdot 12$. Man definiert deshalb allgemein den **Erwartungswert** $E(X)$ einer diskreten Zufallsvariablen X mit Werten Ω' als

$$E(X) := \sum_{x \in \Omega'} p(x)x = \sum_{x \in \Omega'} P(X = x)x = \sum_{\omega \in \Omega} P(\{\omega\})X(\omega) \ . \tag{A.2}$$

Statt erwartungswert spricht man gleichbedeutend auch vom Mittelwert oder Durchschnittswert.

[2] Um $p(x)$ zu bestimmen muss man einfach zählen wieviele Elementarereignisse (n_1, n_2) die Augenzahl $X = x$ ergeben. Z.B. folgt Augenzahl $x = 2$ nur für $(1, 1)$; $x = 3$ für $(1, 2), (2, 1)$; $x = 4$ für $(1, 3), (2, 2), (3, 1)$;...;$x = 7$ für $(1, 6), (2, 5), \ldots, (6, 1)$; $x = 8$ für $(2, 6), (3, 5), \ldots, (6, 2)$;...;$x = 12$ nur für $(6, 6)$.

Beispiele: a) Die Zufallsvariable X mit Verteilung $p(x)$ nach obiger Tabelle hat den Erwartungswert $E(X) = \sum_{x=2}^{12} p(x)x = \frac{1}{36}(2+12)+\frac{2}{36}(3+11)+\ldots+\frac{5}{36}(6+8)+\frac{6}{36}\cdot 7 = \frac{1+2+3+4+5+3}{36}\cdot 14 = \frac{14}{2} = 7$.
b) Die Zufallsvariable Y aus obigem Beispiel mit Verteilung $p(x) = 1/6$ hat die Erwartung $E(Y) = \frac{1}{6}(1+2+\ldots+6) = 3.5$.
c) Dasselbe gilt auch für die Zufallsvariable Z, d.h. $E(Z) = 3.5$.

Der Erwartungswert ist linear: Gegeben seien zwei Zufallsvariablen $X : \Omega \to \Omega'$ und $Y : \Omega \to \Omega'$ mit Verteilungen $p_X(x)$ und $p_Y(x)$ sowie Erwartungswerten $E(X)$ und $E(Y)$. Dann ergibt sich für Konstanten a, b der Erwartungswert der gewichteten Summe $aX + bY$ als

$$E(aX + bY) = aE(X) + bE(Y) \,. \tag{A.3}$$

Beweis: $E(aX + bY) := \sum_{\omega\in\Omega} P(\{\omega\})(aX(\omega) + bY(\omega)) = a\sum_{\omega\in\Omega} P(\{\omega\})X(\omega) + b\sum_{\omega\in\Omega} P(\{\omega\})Y(\omega)) = aE(X) + bE(Y)$. □
Beispiel: Für obige Beispiele gilt offenbar $X = Y + Z$. Deshalb ist $E(X) = E(Y + Z) = E(Y) + E(Z) = 3.5 + 3.5 = 7$, was mit der (komplizierteren) obigen Rechnung übereinstimmt.

Neben dem Mittelwert interessiert man sich für die mittlere Abweichung einer Zufallsvariablen von ihrem Mittelwert. Dazu definiert man die **Varianz** als das Quadrat dieser mittleren Abweichung,

$$\mathrm{Var}(X) := E((X - E(X))^2) = E(X^2) - E^2(X) \geq 0 \tag{A.4}$$

Beweis (für die Gleichung): Wegen der 2.Binomischen Formel und der Linearität des Erwartungswerts folgt $E((X - E(X))^2) = E(X^2 - 2XE(X) + E^2(X)) = E(X^2) - 2E(X)E(X) + E^2(X)$. □
Offenbar gilt

$$\mathrm{Var}(X) = 0 \qquad \text{genau dann, wenn} \qquad X = E(X) \ \text{ konstant ist.} \tag{A.5}$$

Beweis: Sei $p(x)$ die Verteilung von X. Nach (A.4) mit (A.2) gilt dann $\mathrm{Var}(X) = \sum_x p(x)(x-E(X))^2$. Da $\mathrm{Var}(X)$ also eine Summe nicht-negativer Summanden $p(x)(x-E(X))^2 \geq 0$ ist gilt $\mathrm{Var}(X) = 0$ genau dann, wenn $p(x)(x - E(X))^2 = 0$ für alle x. Dies gilt genau dann, falls $p(x) = 0$ oder $x = E(X)$ für alle x gilt. Dies gilt genau dann, falls $x = E(X)$ für alle x deren Wahrscheinlichkeit $p(x) > 0$ ist, d.h. falls $X = E(X)$ eine konstante Zufallsvariable ist. □
Die Wurzel der Varianz nennt man auch Standardabweichung oder Streuung (engl. *standard deviation*) der Zufallsvariablen X,

$$\mathrm{SD}(X) := \sqrt{\mathrm{Var}(X)} \,. \tag{A.6}$$

Die Varianz ist nicht linear(!). Stattdessen gilt für Konstanten a, c

$$\mathrm{Var}(aX + c) = a^2\mathrm{Var}(X) \qquad \text{bzw.} \qquad \mathrm{SD}(aX + c) = a\mathrm{SD}(X) \tag{A.7}$$

Beweis: $\mathrm{Var}(aX+c) = E((aX+c)^2) - E^2(aX+c) = E(a^2X^2+2acX+c^2) - (aE(x)+c)^2 = a^2E(X^2) + 2acE(X) + c^2 - (a^2E^2(x) + 2acE(X) + c^2) = a^2(E(X^2) - E^2(x)) = a^2\mathrm{Var}(X)$. Damit gilt $\mathrm{SD}(aX+c) = \sqrt{\mathrm{Var}(aX+c)} = \sqrt{a^2\mathrm{Var}(X)} = a\mathrm{SD}(X)$. □
Beispiel: Für die Zufallsvariable $Y(n_1, n_2) = n_1$ aus obigen Beispielen (Wurf mit zwei Würfeln) kennen wir schon E(Y)=3.5. Außerdem ist $E(Y^2) = \frac{1}{6}(1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 6^2) = \frac{1+4+9+16+25+36}{6} = \frac{91}{6} \approx 15.2$. Damit ergibt sich die Varianz nach (A.4) als $\mathrm{Var}(Y) = E(Y^2) - E^2(Y) = \frac{91}{6} - 3.5^2 = \frac{35}{12} \approx 2.92$ und die Standardabweichung als $\mathrm{SD}(Y) = \sqrt{\mathrm{Var}(Y)} \approx 1.71$.

A.5 Mehrere Zufallsvariablen: Gemeinsame Verteilung, Bedingte Verteilung, Unabhängigkeit, Kovarianz

Gegeben seien zwei (oder mehr) Zufallsvariablen X und Y. Die **gemeinsame Verteilung** p von X und Y definiert man als

$$p(x,y) := P(X = x \text{ und } Y = y) = P(\{\omega \in \Omega : X(\omega) = x \text{ und } Y(\omega) = y) \ . \qquad \text{(A.8)}$$

Bemerkung: Diese Definition ist eigentlich nichts wirklich neues, denn man kann das Tupel (x, y) auch zu einem Vektor $\mathbf{v} := (x, y)$ zusammenfassen. Ähnlich wird aus den beiden Zufallsvariablen (X, Y) ein Zufallsvektor $\mathbf{V} := (X, Y)$, der ebenfalls höchstens abzählbar viele Werte annehmen kann, und deshalb eigentlich auch nichts anderes als eine "normale" Zufallsvariable (wie bisher betrachtet) ist. D.h. alle bisherigen Betrachtungen (z.B. über Erwartungswert, Varianz, etc.) gelten genauso auch für gemeinsame Verteilungen und Zufallsvektoren.
Beispiel: Die gemeinsame Verteilung der oben definierten Zufallsvariablen $Y(n_1, n_2) = n_1$ und $Z(n_1, n_2) = n_2$ ist $p(y, z) := P(Y = y, Z = z) = 1/36$.

Gegeben seien zwei Ereignisse $A, B \in \Sigma$. Dann definiert man die **bedingte Wahrscheinlichkeit** von A gegeben B als

$$P(A|B) := \frac{P(A \cap B)}{P(B)} \ . \qquad \text{(A.9)}$$

Man beachte, dass für jede feste Bedingung $B \neq \varnothing$ durch $P(A|B)$ ein neuer Wahrscheinlichkeitsraum $(\Omega^*, \Sigma^*, P^*)$ für die auf B eingeschränkte Ergebnismenge $\Omega^* := B \subseteq \Omega$ bzw. Ereignismenge $\Sigma^* := \{\sigma \in \Sigma : \sigma \subseteq B\}$ definiert wird: Denn das Wahrscheinlichkeitsmaß $P^*(A^*) := \frac{P(A^*)}{P(B)}$ erfüllt offenbar alle Kolmogoroff-Axiome. Insbesondere re-normalisiert das Teilen durch $P(B)$ das neue Maß auf $P^*(\Omega^*) = P^*(B) = 1$. Umformen von (A.9) ergibt allgemein

$$P(A \cap B) = P(B)P(A|B) \qquad (= P(A)P(B|A)) \ . \qquad \text{(A.10)}$$

Aus (A.9) und (A.10) folgt unmittelbar das **Bayes'sche Theorem**, mit welchem man das zu messende Ereignis A und die Bedingung B vertauschen kann,

$$P(A|B) = P(B|A)\frac{P(A)}{P(B)} \ . \tag{A.11}$$

Beweis: $P(A|B) \overset{(A.9)}{:=} \frac{P(A\cap B)}{P(B)} \overset{(A.10)}{=} \frac{p(A)P(B|A)}{p(B)}$. □

Man nennt nun zwei Ereignisse **unabhängig**, falls

$$P(A \cap B) = P(B)P(A) \qquad \text{bzw.} \qquad P(A|B) = P(A) \tag{A.12}$$

d.h. falls sich die Wahrscheinlichkeit $P(A)$ von Ereignis A nicht ändert wenn das Ereignis B beobachtet wird.

Beispiele: a) Seien $A := \{(1,1),(1,2),(2,1),(2,2)\}$ ="*Augenzahl höchstens 2 pro Würfel*" und $B := \{(1,1),(2,2),(3,3)\}$ ="*1er-,2er- oder 3er-Pasch*" zwei Ereignisse beim Würfeln mit zwei Würfeln mit $\Omega := \{(n_1,n_2)|n_1,n_2 \in \{1,\ldots,6\}\}$ aus obigen Beispielen. Dann gilt $P(A|B) = \frac{P(\{(1,1),(2,2)\})}{P(\{(1,1),(2,2),(3,3)\})} = \frac{2/36}{3/36} = 2/3$. Ähnlich gilt $P(B|A) = \frac{P(\{(1,1),(2,2)\})}{P(\{(1,1),(1,2),(2,1),(2,2)\})} = \frac{2/36}{4/36} = 0.5$.
b) Die Ereignisse $C := \{(3,n_2)|n_2 \in \{1,\ldots,6\}\}$ ="*Erster Würfel hat Augenzahl 3*" und $D := \{(n_1,6)|n_2 \in \{1,\ldots,6\}\}$ ="*Zweiter Würfel hat Augenzahl 6*" sind offenbar unabhängig, denn wegen $P(C) = P(D) = 1/6$ gilt $P(C \cap D) = P(\{(3,6)\}) = 1/36 = P(C) \cdot P(D)$ bzw. äquivalent $P(C|D) = \frac{1/36}{1/6} = \frac{1}{6} = P(C)$.

Aus (A.9) und (A.12) ergeben sich sofort **entsprechende Definitionen für Zufallsvariablen** $X : \Omega \to \Omega'$ und $Y : \Omega \to \Omega''$: Die bedingte Verteilung $p(X|\,Y)$ von X gegeben den Wert von Y ist

$$P(X = x|\,Y = y) := \frac{P(X = x,\, Y = y)}{P(\,Y = y)} = \frac{P(\{\omega \in \Omega : X(\omega) = x \wedge Y(\omega) = y\})}{P(\{\omega \in \Omega :\, Y(\omega) = y\})} \ . \tag{A.13}$$

Da für festes $Y = y$ die Summe $\sum_{x\in\Omega'} P(X = x|\,Y = y) = 1$ wieder auf 1 normiert ist (andernfalls wäre $P(X|\,Y)$ ja keine Verteilung), ergibt sich nach beidseitigem Multiplizieren von $P(\,Y = y)$ der Satz der totalen Wahrscheinlichkeit,

$$P(\,Y = y) = \sum_{x\in\Omega'} P(X = x|\,Y = y)P(\,Y = y) = \sum_{x\in\Omega'} P(X = x,\, Y = y) \ . \tag{A.14}$$

Selbstverständlich gilt (mit denselben Umformungen wie für (A.11)) das Bayes'sche Theorem auch für Zufallsvariablen,

$$P(\,Y = y|X = x) = P(X = x|\,Y = y)\frac{P(\,Y = y)}{P(X = x)} \ . \tag{A.15}$$

Man nennt zwei Zufallsvariablen unabhängig, falls für alle $x \in \Omega', y \in \Omega''$

$$P(X=x, Y=y) = P(X=x)P(Y=y) \qquad \text{bzw.} \qquad P(X=x|\,Y=y) = P(X=x) \tag{A.16}$$

gilt. Falls X und Y unabhängige Zufallsvariablen sind, dann sind offensichtlich auch beliebige Wertemengen A,B von X, Y unabhängige Ereignisse, sowie beliebige Funktionen $f(X)$ und $g(Y)$ von X, Y unabhängige Zufallsvariablen:

$$\text{(A.16)} \quad \Rightarrow \quad P(X \in A, Y \in B) = P(X \in A)P(Y \in B) \quad \forall A, B \tag{A.17}$$

$$\text{(A.16)} \quad \Rightarrow \quad P(f(X)=x', g(Y)=y') = P(f(X)=x')P(g(Y)=y') \quad \forall x', y' \tag{A.18}$$

Beweis: (A.17) folgt mit Kolmogoroff-Axiom III von Seite 213: $P(X \in A, Y \in B) = \sum_{x\in A}\sum_{y\in B} P(X=x, Y=y) = \sum_{x\in A}\sum_{y\in B} P(X=x)P(Y=y) = \sum_{x\in A} P(X=x)\sum_{y\in B} P(Y=y) = P(X \in A)P(Y \in B)$. Damit folgt (A.18) wegen $P(f(X)=x', g(Y)=y') = P(X \in f^{-1}(x'), Y \in g^{-1}(y')) = P(X \in f^{-1}(x'))P(Y \in g^{-1}(y')) = P(f(X)=x')P(g(Y)=y')$, wobei $f^{-1}(x') := \{x|f(x)=x'\}$ und $g^{-1}(y') := \{y|g(y)=y'\}$ die inversen Wertemengen zu x' und y' sind. □

Für unabhängige Zufallsvariablen vereinfachen sich viele Berechnungen enorm. Z.B. gilt für unabhängige X, Y

$$E(XY) = E(X)E(Y) \tag{A.19}$$

$$\mathrm{Var}(X+Y) = \mathrm{Var}(X) + \mathrm{Var}(Y) \tag{A.20}$$

$$\mathrm{Var}(XY) = E(X^2)E(Y^2) - E^2(X)E^2(Y) \tag{A.21}$$

$$= \mathrm{Var}(X)\mathrm{Var}(Y) + E^2(X)\mathrm{Var}(Y) + \mathrm{Var}(X)E^2(Y)\ . \tag{A.22}$$

Beweis: $E(XY) := \sum_x\sum_y p(x,y)xy = \sum_x\sum_y p(x)p(y)xy = \sum_x p(x)x\sum_y p(y)y = E(X)E(Y)$ und damit auch $\mathrm{Var}(X+Y) := E((X+Y)^2) - E^2(X+Y) = E(X^2+2XY+Y^2) - E^2(X+Y) = E(X^2) + 2E(X)E(Y) + E(Y^2) - E^2(X) - E^2(Y) - 2E(X)E(Y) = E(X^2) - E^2(X) + E(Y^2) - E^2(Y) = \mathrm{Var}(X) + \mathrm{Var}(Y)$, womit (A.19) und (A.20) gezeigt sind. Damit folgt (A.21) wegen $\mathrm{Var}(XY) := E((XY)^2) - E^2(XY) = E(X^2Y^2) - (E(X)E(Y))^2 \overset{(A.18,A.19)}{=} E(X^2)E(Y^2) - E^2(X)E^2(Y)$. Das ist dasselbe wie (A.22) wegen $\mathrm{Var}(X)\mathrm{Var}(Y) + E^2(X)\mathrm{Var}(Y) + \mathrm{Var}(X)E^2(Y) := (E(X^2) - E^2(X))(E(Y^2) - E^2(Y)) + E^2(X)(E(Y^2) - E^2(Y)) + (E(X^2) - E^2(X))E^2(Y) = E(X^2)E(Y^2) - E^2(X)E(Y^2) - E(X^2)E^2(Y) + E^2(X)E^2(Y) + E^2(X)E(Y^2) - E^2(X)E^2(Y) + E(X^2)E^2(Y) - E^2(X)E^2(Y) = E(X^2)E(Y^2) - E^2(X)E^2(Y)$. □

Beispiele: Für $X(n_1,n_2) = n_1 + n_2$, $Y(n_1,n_2) = n_1$ und $Z(n_1,n_2) = n_2$ wie in obigen Beispielen (Wurf mit zwei Würfeln) sowie $W(n_1,n_2) = n_1 \cdot n_2$ gilt:

a) $P(Y=3|Z=6) = \frac{P(Y=3,Z=6)}{P(Z=6)} = \frac{1/36}{1/6} = \frac{1}{6}$ nach (A.13) wie in obigem Beispiel.

b) Da für beliebige $y, z \in \{1,\ldots,6\}$ stets $P(Y=y|Z=z) = \frac{P(Y=y,Z=z)}{P(Z=z)} = \frac{1/36}{1/6} = \frac{1}{6} =$

$P(Y = y)$ gilt, sind Y und Z nach (A.16) unabhängig.
c) Für das Produkt $W = Y \cdot Z$ der Augenzahlen der beiden Würfel folgt wegen der Unabhängigkeit von Y und Z aus (A.19) der Erwartungswert $E(W) = E(YZ) = E(Y)E(Z) = 3.5 \cdot 3.5 = 12.25$.
d) Mit $E(Y) = E(Z) = 7/2$ und $\mathrm{Var}(Y) = \mathrm{Var}(Z) = \frac{35}{12}$ (siehe oben) folgt wegen der Unabhängigkeit von Y und Z für die Augen-Summe $X = Y + Z$ nach (A.20) die Varianz $\mathrm{Var}(X) = \mathrm{Var}(Y) + \mathrm{Var}(Z) = \frac{35}{12} + \frac{35}{12} = \frac{35}{6} \approx 5.83$ und damit die Standardabweichung $\mathrm{SD}(X) = \sqrt{\mathrm{Var}(X)} \approx 2.41$.
e) **Gesetz der großen Zahlen:** Sei $X_1, X_2, \ldots, X_n$ eine Folge unabhängiger Zufallsvariablen mit identischer Verteilung $P(X_1 = x) = P(X_2 = x) = \ldots = P(X_n = x)$ für alle x, so dass insbesondere alle Zufallsvariablen denselben Erwartungswert $\mu := E(X_1) = \ldots = E(X_n)$ haben. Dann gilt

$$\mu = \lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} X_i \tag{A.23}$$

Beweis: Sei $\sigma^2 := \mathrm{Var}(X_1) = \ldots = \mathrm{Var}(X_n)$ die Varianz der Zufallsvariablen und σ entsprechend die Standardabweichung. Wegen der Unabhängigkeit der X_i folgt aus (A.20) dann $\mathrm{Var}(\sum_{i=1}^n X_i) = \mathrm{Var}(X_1) + \ldots + \mathrm{Var}(X_n) = n\sigma^2$. Aus (A.7) folgt damit $\mathrm{Var}(\frac{1}{n}\sum_{i=1}^n X_i) = \frac{1}{n^2}\mathrm{Var}(\sum_{i=1}^n X_i) = \frac{1}{n^2} \cdot n\sigma^2 = \frac{\sigma^2}{n} \to 0$ für $n \to \infty$. Da die Varianz 0 wird muss für $n \to \infty$ die Zufallsvariable $\frac{1}{n}\sum_{i=1}^n X_i$ wegen (A.5) identisch zu ihrem Erwartungswert $E(\frac{1}{n}\sum_{i=1}^n X_i) = \frac{1}{n}\sum_{i=1}^n E(X_i) = \frac{1}{n} \cdot n\mu = \mu$ werden. □

Schließlich definieren wir noch die **Kovarianz** zweier Zufallsvariablen X, Y als

$$\mathrm{Cov}(X, Y) := E\left[(X - E(X))(Y - E(Y))\right] = E(XY) - E(X)E(Y) \tag{A.24}$$

Beweis (für die Gleichung): Wegen (A.3) folgt $E\left[(X - E(X))(Y - E(Y))\right] = E\left[XY - E(X)\,Y - XE(Y) + E(X)E(Y)\right] = E(XY) - E(X)E(Y) - E(X)E(Y) + E(X)E(Y) = E(XY) - E(X)E(Y)$. □
Bemerkungen: Die Kovarianz ist ein Mass dafür ob X, Y in dieselbe Richtung variieren: Eine positive Kovarianz $\mathrm{Cov}(X, Y) > 0$ bedeutet, dass falls X groß ist auch Y groß ist (und umgekehrt). Negative $\mathrm{Cov}(X, Y) < 0$ bedeutet, dass falls X groß ist dann Y klein ist (und umgekehrt). Außerdem gilt:

$$\mathrm{Cov}(Y, X) = \mathrm{Cov}(X, Y), \tag{A.25}$$
$$\mathrm{Cov}(X, X) = \mathrm{Var}(X), \tag{A.26}$$
$$\text{Falls } X, Y \text{ unabhängig, dann } \mathrm{Cov}(X, Y) = 0 \tag{A.27}$$

Beweis: Die Symmetrie (A.25) folgt direkt aus der Definition (A.24). (A.26) folgt aus der Definition (A.4) der Varianz. (A.27) folgt wegen (A.19). □
Die Umkehrung von (A.27) gilt leider nicht. D.h. aus $\mathrm{Cov}(X, Y) = 0$ folgt nicht notwendigerweise die Unabhängigkeit von X und Y, wie das folgende Beispiel zeigt.

Beispiel: Wir konstruieren zwei Zufallsvariablen X und Y die voneinander abhängen aber trotzdem Kovarianz Null haben:
Seien hierfür W_1 und W_2 zwei unabhängige Zufallsvariablen mit $E(W_1) = E(W_2) = 0$, und Z_1 und Z_2 zwei weitere beliebige Zufallsvariablen, die unabhängig von W_1, W_2 sind. Wir betrachten die Produkte $X := W_1 Z_1$ und $Y := W_2 Z_2$. Dann gilt allgemein

$$\mathrm{Cov}(X, Y) = \mathrm{Cov}(W_1 Z_1, W_2 Z_2) = 0 \tag{A.28}$$

denn wegen (A.19) und $E(W_1) = E(W_2) = 0$ gilt $\mathrm{Cov}(W_1 Z_1, W_2 Z_2) := E(W_1 Z_1 W_2 Z_2) - E(W_1 Z_1) E(W_2 Z_2) = E(W_1) E(W_2) E(Z_1 Z_2) - E(W_1) E(Z_1) E(W_2) E(W_3) = 0$. Andererseits sind X und Y im Allgemeinen abhängig falls Z_1 und Z_2 voneinander abhängen. Z.B. für binäre $W_1, W_2 \in \{-1, 1\}$ mit $P(W_i = 1) = 0.5$ und $Z_1, Z_2 \in \{0, 1\}$ mit $P(Z_i = 1) = 0.5$ sowie der (deterministischen) Abhängigkeit $Z_2 := 1 - Z_1$ folgt offenbar $P(X = 0) = 0.5$ aber $P(X = 0 | Y = 0) = 0$ was der Definition von Unabhängigkeit nach (A.16) widerspricht. D.h. obwohl $\mathrm{Cov}(X, Y) = 0$ nach (A.28) sind X und Y trotzdem abhängig.

Mit der Definition der Kovarianz (A.24) kann man elegant den allgemeinen **Varianzen-Summensatz** formulieren: Für beliebige Zufallsvariablen X und Y bzw. $X_1, X_2, \ldots, X_n$ gilt

$$\mathrm{Var}(X + Y) = \mathrm{Var}(X) + \mathrm{Var}(Y) + 2\mathrm{Cov}(X, Y) \qquad \text{bzw.} \tag{A.29}$$

$$\mathrm{Var}\left(\sum_{i=1}^{n} X_i\right) = \sum_{i=1}^{n} \mathrm{Var}(X_i) + \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} \mathrm{Cov}(X_i, X_j) \; . \tag{A.30}$$

Beweis: Mit Definition der Varianz (A.4), Linearität des Erwartungswerts (A.3) und binomischer Formel (siehe Skript Mathe-1) folgt (A.29) wegen $\mathrm{Var}(X + Y) := E((X + Y)^2) - E^2(X + Y) = E(X^2 + 2XY + Y^2) - (E(X) + E(Y))^2 = E(X^2) + 2E(XY) + E(Y^2) - E^2(X) - 2E(X)E(Y) - E^2(Y) = \mathrm{Var}(X) + \mathrm{Var}(Y) + 2\mathrm{Cov}(X, Y)$. Ähnlich folgt auch die allgemeine Formel (A.30) wegen $\mathrm{Var}(\sum_{i=1}^n X_i) := E((\sum_i X_i)^2) - E^2(\sum_i X_i) = E((\sum_i X_i)(\sum_j X_j)) - (\sum_i E(X_i))^2 = E(\sum_i \sum_j X_i X_j) - \sum_i \sum_j E(X_i) E(X_j) = \sum_i \sum_j E(X_i X_j) - E(X_i) E(X_j) = \sum_i \sum_j \mathrm{Cov}(X_i, X_j) = \sum_i \mathrm{Cov}(X_i, X_i) + \sum_i \sum_{j \neq i} \mathrm{Cov}(X_i, X_j)$ mit (A.26). □

Bemerkungen: (A.29) ist offenbar ein Spezialfall von (A.30). Falls X und Y unabhängig sind erhält man aus (A.29) mit (A.27) unmittelbar (A.20). Die entsprechenden allgemeinen **Produktsätze für Erwartungswert und Varianz** sind

$$E(XY) = E(X)E(Y) + \mathrm{Cov}(X, Y) \tag{A.31}$$

$$\begin{aligned} \mathrm{Var}(XY) = {} & \mathrm{Var}(X)\mathrm{Var}(Y) + E^2(X)\mathrm{Var}(Y) + \mathrm{Var}(X)E^2(Y) \\ & + \mathrm{Cov}(X^2, Y^2) - \mathrm{Cov}^2(X, Y) - 2\mathrm{Cov}(X, Y)E(X)E(Y) \end{aligned} \tag{A.32}$$

Beweis: (A.31) erhält man einfach durch Umstellen von (A.24). Damit und mit (A.4) folgt (A.32) wegen $\mathrm{Var}(XY) = E(X^2 Y^2) - E^2(XY) = E(X^2)E(Y^2) + \mathrm{Cov}(X^2, Y^2) -$

$(E(X)E(Y) + \mathrm{Cov}(X,Y))^2 = E(X^2)E(Y^2) - E^2(X)E^2(Y) + \mathrm{Cov}(X^2,Y^2) - \mathrm{Cov}^2(X,Y) - 2E(X)E(Y)\mathrm{Cov}(X,Y)$, da für die ersten drei Terme die Identität

$$E(X^2)E(Y^2) - E^2(X)E^2(Y) = \mathrm{Var}(X)\mathrm{Var}(Y) + E^2(X)\mathrm{Var}(Y) + \mathrm{Var}(X)E^2(Y)$$

gilt, welche bereits im Beweis zu (A.21) und (A.22) gezeigt wurde (ohne Verwenden der Unabhängigkeits-Annahme). □

A.6 Kontinuierliche Zufallsvariablen und Verteilungen: Dichte und Verteilungsfunktionen

Für Kontinuierliche Zufallsvariablen $X : \Omega \to \mathbb{R}$ gelten prinzipiell dieselben Definitionen und Sätze wie in den vorigen Abschnitten. Eine Schwierigkeit bei kontinuierlichen Zufallsvariablen ist dass einem einzelnen Wert $X = x$ im Allgemeinen das Maß $P(X = x) = 0$ zugeordnet werden muss (da es ja überabzählbar viele verschiedene x gibt) und demnach also auch $\sum_x P(X = x) = 0$ ist. Um positive Wahrscheinlichkeiten zu bekommen betrachtet man deshalb immer Intervalle $x \in [a; b] \in \Sigma$ und definiert die Wahrscheinlichkeits-Dichte oder kurz Dichte für $x \in \mathbb{R}$ als

$$p(x) := \lim_{\epsilon \to 0} \frac{P(X \in [x - \epsilon; x + \epsilon])}{2\epsilon} \ . \tag{A.33}$$

D.h. im Gegensatz zur Verteilung diskreter Zufallsvariablen ist hier $p(x) \neq P(x) = 0$! Trotzdem gelten natürlich weiterhin die Kolmogorov-Axiome (siehe Kapitel A.2). Insbesondere muss deshalb für jede disjunkte Zerlegung $\mathbb{R} = I_1 \cup I_2 \cup ... \cup I_n$ der reellen Zahlen in Intervalle $I_i \in \Sigma$ gelten, dass $P(I_1) + P(I_2) + \ldots + P(I_n) = 1$ ist. Für $n \to \infty$ und verschwindende Intervalle mit $\max(I_i) - \min(I_i) \to 0$ wird dann aus der vorigen Summierung zu 1 ein Riemann-Integral (siehe Skript Mathe-I,S. 134,Def. 6.5) mit

$$\int_{-\infty}^{\infty} p(x)dx = 1 \ . \tag{A.34}$$

Beispiel (Gleichverteilung): Man nennt eine Zufallsvariable $X \in \mathbb{R}$ gleichverteilt auf dem Intervall $[a; b]$, falls $P(X \in [c; d]) = \frac{d-c}{b-a}$ gilt. Damit folgt aus (A.33) für $x \in [a, b]$ die Dichte $p(x) = \lim_{\epsilon \to 0} \frac{P(X \in [x-\epsilon; x+\epsilon])}{2\epsilon} = \lim_{\epsilon \to 0} \frac{2\epsilon/(b-a)}{2\epsilon}$, d.h.

$$p(x) = \begin{cases} \frac{1}{b-a} & , x \in [a, b] \\ 0 & , \text{sonst} \end{cases} \tag{A.35}$$

Z.B. ist für eine Gleichverteilung von X auf $[0; 10]$ die Dichte p(x)=1/10, und $P(X \in [2; 7]) = 5/10 = 0.5$.

Als **Verteilungsfunktion** $F_X(x) : \mathbb{R} \to [0,1]$ einer Zufallsvariablen $X : \Omega \to \mathbb{R}$ bezeichnet man die Funktion

$$F_X(x) := P(X \leq x) = \int_{-\infty}^{x} p(x)\,dx \ . \tag{A.36}$$

Beispiel: Die Verteilungsfunktion einer Gleichverteilung mit Dichte (A.35) ist für $x \in [a,b]$

$$F_X(x) = \int_{-\infty}^{x} p(x)\,dx = \int_a^x \frac{1}{b-a}\,dx' = \left[\frac{x'}{b-a}\right]_a^x = \frac{x-a}{b-a} \tag{A.37}$$

sowie $F_X(x) = 0$ für $x \leq a$ und $F_X(x) = 1$ für $x \geq b$.

Den **Erwartungswert einer kontinuierlichen Zufallsvariablen** definiert man ähnlich wie in (A.2): Für eine hinreichend feine Zerlegung $\mathbb{R} = I_1 \cup I_2 \cup ... \cup I_n$ der reellen Achse in n kleine diskunkte Teilintervalle I_i mit Mittelpunkt $\bar{I}_i := \frac{\min(I_i)+\max(I_i)}{2}$ gilt nach (A.2) die Näherung $E(X) = \sum_{i=1}^{n} P(I_i)\bar{I}_i$ als Summe, welche im Limit $n \to \infty$ zum Riemann-Integral [3] wird:

$$E(X) := \int_{-\infty}^{\infty} p(x)x\ dx \tag{A.38}$$

Beispiel: Der Erwartungswert einer Gleichverteilung auf $[a;b]$ ist mit (A.35)

$$\begin{aligned} E(X) &:= \int_{-\infty}^{\infty} p(x)x\ dx = \int_a^b \frac{1}{b-a}x\ dx = \left[\frac{x^2}{2(b-a)}\right]_a^b = \frac{b^2-a^2}{2(b-a)} = \frac{(b-a)(b+a)}{2(b-a)} \\ &= \frac{a+b}{2} \end{aligned} \tag{A.39}$$

Alle anderen Größen (z.B. Varianz, Kovarianz, etc.) werden mit Hilfe von Erwartungswerten definiert, sodass für kontinuierliche Zufallsvariablen keine neuen Definitionen notwendig sind.
Beispiel (Varianz und Standardabweichung einer Gleichverteilung): Falls X

[3] Mit (halben) Intervalllängen ϵ_i ist $I_i = [\bar{I}_i - \epsilon_i; \bar{I}_i + \epsilon_i]$. Mit (A.33) von Seite 221 ist damit $P(I_i) := P(X \in I_i) = 2\epsilon_i p(\bar{I}_i)$ und damit $E(X) = \lim_{n\to\infty} \sum_{i=1}^{n} p(\bar{I}_i)\bar{I}_i \cdot 2\epsilon_i$. Für $n \to \infty$ bilden die $\bar{I}_i$ eine immer feinere Zerlegung der x-Achse, und die Intervallbreite $2\epsilon_i$ wird zum Differenzial dx. Der Rest folgt aus der Definition des Riemann-Integrals (siehe Skript Mathe-I,S. 134,Def. 6.5).

eine auf $[a;b]$ gleichverteilte Zufallsvariable ist, dann ist nach (A.4) und (A.6)

$$E(X^2) := \int_{-\infty}^{\infty} p(x)x^2 \; dx = \int_a^b \frac{1}{b-a}x^2 dx = \left[\frac{x^3}{3(b-a)}\right]_a^b = \frac{b^3-a^3}{3(b-a)} \tag{A.40}$$

$$\begin{aligned}\mathrm{Var}(X) &:= E(X^2) - E^2(X) = \frac{b^3-a^3}{3(b-a)} - \left(\frac{a+b}{2}\right)^2 \\ &= \frac{4b^3-4a^3-3(b-a)(a^2+2ab+b^2)}{12(b-a)} \\ &= \frac{4b^3-4a^3+3a^3+6a^2b+3ab^2-3a^2b-6ab^2-3b^3}{12(b-a)} \\ &= \frac{b^3-3ab^2+3a^2b-a^3}{12(b-a)} = \frac{(b-a)^3}{12(b-a)} = \frac{(b-a)^2}{12}\end{aligned} \tag{A.41}$$

$$\mathrm{SD}(X) := \sqrt{\mathrm{Var}(X)} = \frac{b-a}{\sqrt{12}} \tag{A.42}$$

Auch die Definitionen der *gemeinsamen Verteilung* (A.8), *bedingten Verteilung* (A.13), *Unabhängigkeit* (A.16), *Kovarianz* (A.24) sowie die daraus hergeleiteten Rechengesetze (A.19), (A.20), (A.26), (A.27) übertragen sich offenbar unverändert auf kontinuierliche Zufallsvariablen, indem man ggf. die diskrete Wahrscheinlichkeitsverteilung $p(.)$ durch eine entsprechende Dichtefunktion $p(.)$ ersetzt. Abschließend betrachten wir zwei wichtige Verteilungen etwas genauer: Die Binomialverteilung und die Gauß-Verteilung...

A.7 Bernoulli-Verteilung und Binomialverteilung

Bernoulli-Verteilung: Die Verteilung einer binären Zufallsvariable $\beta \in \{0,1\}$ nennt man Bernoulli-verteilt. Falls $p := P(\beta = 1)$, dann folgt offenbar $P(\beta = 0) = 1 - p$. Erwartung, Varianz und Standardabweichung ergeben sich dann aus (A.2),(A.4),(A.6) als

$$E(\beta) := p \cdot 1 + (1-p) \cdot 0 = p \tag{A.43}$$

$$\mathrm{Var}(\beta) := E(\beta^2) - E^2(\beta) = E(\beta) - E^2(\beta) = p - p^2 = p(1-p) \tag{A.44}$$

$$\mathrm{SD}(\beta) := \sqrt{\mathrm{Var}(\beta)} = \sqrt{p(1-p)} \tag{A.45}$$

Beispiel: Beim Werfen einer Münze gewinne man bei Kopf einen Euro ($\beta = 1$), währen man bei Zahl nichts gewinnt ($\beta = 0$). Bei einer "gerechten" Münze ist $p = 0.5$. Daraus ergibt sich der erwartete Gewinn $E(\beta) = 0.5$ mit Varianz $\mathrm{Var}(\beta) = 0.5(1-0.5) = 0.25$ und Standardabweichung $\mathrm{SD}(\beta) = \sqrt{0.25} = 0.5$. Falls der Gewinn $G = 5\text{Euro} = 5\beta$ (statt 1 Euro) wäre so gilt mit (A.3) und (A.7) entsprechend $E(5\beta) = 5 \cdot 0.5 = 2.5$, $\mathrm{Var}(5\beta) = 5^2 \cdot 0.25 = 6.25$, und $\mathrm{SD}(5\beta) = 5 \cdot 0.5 = 2.5$.

Binomial-Verteilung: Die Summe $B := \sum_{i=1}^{n} \beta_i$ aus n unabhängigen identisch Bernoulli-verteilten Zufallsvariablen β_i mit $P(\beta_i = 1) = p$ nennt man binomialverteilt mit Parametern n und p. Offenbar ist $B \in \{0, 1, 2, \ldots, n\}$ eine diskrete Zufallsvariable mit Wahrscheinlichkeitsverteilung

$$p_B(k; n, p) := P(B = k) = \binom{n}{k} p^k (1-p)^{n-k} \ . \tag{A.46}$$

Beweis: Die Wahrscheinlichkeit $P(\beta_1, \ldots, \beta_n)$ jeder Bernoulli-Sequenz $\beta_1, \ldots, \beta_n$ mit k Einsen und $n-k$ Nullen ist $p^k(1-p)^{n-k}$. Da es genau $\binom{n}{k}$ solche Sequenzen gibt [4] folgt (A.46). □

Da B über *unabhängige* Bernoulli-Variablen summiert, folgt einfach

$$E(B) = np \ , \qquad \mathrm{Var}(B) = np(1-p) \ , \qquad \mathrm{SD(B)} = \sqrt{np(1-p)} \ . \tag{A.47}$$

Beweis: Aus (A.3) mit (A.43) folgt $E(B) = E(\sum_{i=1}^{n} \beta_i) = \sum_{i=1}^{n} E(\beta_i) = \sum_{i=1}^{n} p = np$. Aus (A.20) mit (A.44) folgt $\mathrm{Var}(B) = \mathrm{Var}(\sum_{i=1}^{n} \beta_i) = \sum_{i=1}^{n} \mathrm{Var}(\beta_i) = \sum_{i=1}^{n} p(1-p) = np(1-p)$. Der Rest folgt aus (A.6). □

Beispiel: Sie werfen $n = 6$ mal eine Münze ($p = 0.5$). Bei Kopf gewinnen Sie jeweils 5 Euro. Was ist erwarteter Gewinn, Varianz und Standardabweichung? Wie groß ist die Wahrscheinlichkeit mindestens 25 Euro zu gewinnen?

Der Gewinn sei $G = 5B$. Damit $E(G) = 5E(B) = 5np = 15$, $\mathrm{Var}(G) = 5^2\mathrm{Var}(B) = 25np(1-p) = 25 \cdot 5 \cdot 0.25 = 125/4 \approx 31.25$, $\mathrm{SD}(G) = \sqrt{31.25} \approx 5.59$. Die Wahrscheinlichkeit 25 Euro oder mehr zu gewinnen ist $P(G \geq 25) = P(B \geq 5) = P(B = 5) + P(B = 6) = \binom{6}{5}0.5^6 + \binom{6}{6}0.5^6 = (6+1)/64 = \frac{7}{64} \approx 0.11$.

A.8 Gauß-Verteilung

Viele Zufallsgrößen aus der Natur und Technik streuen symmetrisch um einen Mittelwert, wobei die Wahrscheinlich stark mit dem Abstand zum Mittelwert abnimmt. Oftmals fällt bei solchen Zufallsvariablen die Dichte exponentiell mit dem Quadrat des Abstands zum Mittelwert, $p(x) \sim \exp(-c(x-\mu)^2)$. Wir wollen zunächst den einfachsten Fall $p(x) \sim e^{-x^2}$ betrachten: Hier gilt offenbar die Symmetrie $p(x) = p(-x)$. Um eine gültige Dichtefunktion mit (A.34) zu erhalten muss das Integral über $p(x)$ genau 1 ergeben. Man interessiert sich deshalb für den Wert

$$I := \int_{-\infty}^{\infty} e^{-x^2} dx = \sqrt{\pi} \tag{A.48}$$

[4] Die Anzahl der Sequenzen β_i mit $\sum_i \beta_i = k$ ergibt sich aus der Anzahl der Möglichkeiten wie man k Einsen aus den n Bernoulli-Variablen β_i auswählen kann, also $\binom{n}{k}$ (vgl. Tabelle/Satz über Kombinatorik auf Seite 211).

Beweis: $I^2 = (2\int_0^\infty e^{-x^2}dx)^2 = 4\int_0^\infty e^{-x^2}dx\int_0^\infty e^{-y^2}dy = 4\int_0^\infty\int_0^\infty e^{-(x^2+y^2)}dx\,dy$, da man mit geschachtelten Integralen genauso wie mit Summen rechnen darf (Distributivgesetz, etc.). Mit der Substitution $y = xs$ und $dy = xds$, sodass y und s wegen $x \geq 0$ dieselben Integrationsgrenzen 0 und ∞ haben, folgt $I^2 = 4\int_0^\infty\int_0^\infty e^{-(x^2+x^2s^2)}dx\,xds = 4\int_0^\infty\left(\int_0^\infty xe^{-x^2(1+s^2)}dx\right)ds$. Für festes s ist der Wert des inneren Integrals $\int_0^\infty xe^{-x^2(1+s^2)}dx = [\frac{e^{-x^2(1+s^2)}}{-2(1+s^2)}]_0^\infty = \frac{1}{2(1+s^2)}$. Mit der Stammfunktion $\int\frac{1}{1+s^2}ds = \arctan(s)$ (siehe Skript Mathe-I, Übungsblatt 12, Aufg. 4f) folgt damit $I^2 = 4\int_0^\infty\frac{1}{2(1+s^2)}ds = 2[\arctan s]_0^\infty = 2(\frac{\pi}{2}-0) = \pi$, und somit $I = \sqrt{\pi}$. □

Somit ist $p(x) = \frac{1}{\sqrt{\pi}}e^{-x^2}$ eine gültige Dichtefunktion. Eine Zufallsvariable mit dieser Dichte $p(x)$ hat Erwartung und Varianz

$$E(X) = \int_{-\infty}^{\infty}\frac{xe^{-x^2}}{\sqrt{\pi}}dx = 0 \quad \text{bzw.} \quad \mathrm{Var}(X) = \int_{-\infty}^{\infty}\frac{x^2e^{-x^2}}{\sqrt{\pi}}dx = \frac{1}{2} \tag{A.49}$$

Beweis: Wegen der Symmetrie $p(x) = p(-x)$ gilt nach (A.38) für den Erwartungswert $E(X) = \int_{-\infty}^{\infty}p(x)x\;dx = \int_{-\infty}^{0}p(x)x\;dx + \int_0^{\infty}p(x)x\;dx = -\int_0^{-\infty}p(x)x\;dx + \int_0^{\infty}p(x)x\;dx = 0$. Damit folgt aus (A.4) die Varianz $\mathrm{Var}(X) = E(X^2) - E^2(X) = E(X^2) = \frac{1}{\sqrt{\pi}}\int_{-\infty}^{\infty}x^2e^{-x^2}dx$. Mit partieller Integration (siehe Skript Mathe-I, S. 140, Satz 6.11) ergibt das Integral $\int_{-\infty}^{\infty}x\cdot(xe^{-x^2})dx = [x\frac{-e^{-x^2}}{2}]_{-\infty}^{\infty} - \int_{-\infty}^{\infty}\frac{-e^{-x^2}}{2}dx = 0 + \frac{\sqrt{\pi}}{2}$, und deshalb $\mathrm{Var}(X) = \frac{1}{\sqrt{\pi}}\frac{\sqrt{\pi}}{2} = \frac{1}{2}$. □

Standard-Normalverteilung: Die mit Faktor $\sqrt{2}$ skalierte Zufallsvariable $N := \sqrt{2}\cdot X$ hat dann nach (A.3) und (A.7) die Erwartung $E(N) = 0$ und Varianz $\mathrm{Var}(N) = \mathrm{SD}(N) = 1$ nach (A.3) und (A.7). Die Verteilung von N nennt man auch Standard-Normalverteilung. (Standardisierte Normalverteilung)

$\mathrm{Var}(c\cdot X) = c^2\cdot\mathrm{Var}(X) \overset{!}{=} 1$
$c^2 = \frac{1}{\mathrm{Var}(X)}$
$c = \frac{1}{\sqrt{\mathrm{Var}(X)}} = \frac{1}{\mathrm{SD}(X)}$
$\mathrm{Var}(X) = \frac{1}{2}$
$\Rightarrow c = \frac{1}{\sqrt{\frac{1}{2}}} = \sqrt{2}$

Gauß-Verteilung: Skaliert man N mit Faktor σ und verschiebt dann um μ, dann nennt man die resultierende Zufallsvariable $G := \sigma\cdot N + \mu$ allgemein Gauß-verteilt. Ihre Dichte, Erwartung, Varianz und Standardabweichung sind

⇒ Dichte der Gauß-Verteilung
Gibt Glocke maximum liegt bei μ
σ ist Standardabweichung / Breite der Glocke

$$p_G(x) = \mathcal{N}(x;\mu,\sigma^2) := \frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

$$E(G) = \mu\;, \qquad \mathrm{Var}(G) = \sigma^2\;, \qquad \mathrm{SD}(G) = \sigma\;. \tag{A.50}$$

Beweis: Nach (A.3) $E(G) = E(\sigma\cdot N+\mu) = \sigma E(N)+\mu = \mu$. Nach (A.7) gilt $\mathrm{Var}(G) = \mathrm{Var}(\sigma\cdot N+\mu) = \sigma^2\mathrm{Var}(N) = \sigma^2$ und damit $\mathrm{SD}(G) = \sqrt{\mathrm{Var}(G)} = \sigma$. Die Dichte erhält man über $G = \sigma N+\mu = \sqrt{2}\sigma X+\mu$. Nach Umformen ist wegen $X = \frac{G-\mu}{\sqrt{2\sigma^2}}$ für $\epsilon\to 0$ dann $P(G\in[x-\epsilon;x+\epsilon]) = P(X\in[\frac{x-\epsilon-\mu}{\sqrt{2\sigma^2}};\frac{x+\epsilon-\mu}{\sqrt{2\sigma^2}}$. D.h. mit $x' := \frac{x-\mu}{\sqrt{2\sigma^2}}$ gilt $P(G\in[x-\epsilon;x+\epsilon]) = P(X\in[x'-\frac{\epsilon}{\sqrt{2\sigma^2}};x'+\frac{\epsilon}{\sqrt{2\sigma^2}}]) = p(x')\frac{2\epsilon}{\sqrt{2\sigma^2}}$. Damit folgt die Dichte $p_G(x) = \frac{P(G\in[x-\epsilon;x+\epsilon])}{2\epsilon} = \frac{p(x')}{\sqrt{2\sigma^2}}$. Nach Einsetzen der Dichte $p(.)$ von X wie über (A.49) definiert folgt damit die behauptete Dichte $p_G(x)$. □

Bemerkungen:

- Die Standard-Normal-Verteilung ist also eine Gauß-Verteilung mit Erwartung $\mu = 0$ und Varianz $\sigma^2 = 1$.
- **Zentraler Grenzwertsatz** (ohne Beweis): Gegeben sei die Summe $S_n := X_1 + X_2 + \ldots + X_n$ eine Folge von n unabhängiger identisch verteilter Zufallsvariablen $X_1, X_2, \ldots, X_n$ mit $\mu := E(X_i)$ und $\sigma := \mathrm{SD}(X_i)$. Dann konvergiert für $n \to \infty$ die standardisierte Summe $\frac{S_n - n\mu}{\sigma\sqrt{n}}$ stets gegen die Standardnormalverteilung. → Addiv ist alles möglich so verteilt!
- Der Zentrale Grenzwertsatz ist der Grund warum viele Größen in Natur, Technik und Wirtschaft Gauß-verteilt sind. Denn dort ergeben sich viele Größen aus der Summation weiterer Größen die ähnlich verteilt sind.
  Beispiele: a) Wirtschaft: Der Jahres-Umsatz einer Firma ergibt sich aus der Summe vieler Verkäufe an einzelne Kunden, und folgt deshalb einer Gauß-Verteilung.
  b) Natur: Die Niederschlagsmenge an einem Ort pro Jahr ergibt sich aus der Summe der Niederschlagsmenge an jedem Tag.
  c) Technik: Bedarf an Stromleistung eines Kraftwerks ergibt sich als Summe der Stromleistungen der vielen Verbraucher. Und so weiter...
- Gauß-Verteilungen spielen eine sehr wichtige Rolle in probabilistischen Modellen für maschinelles Lernen, da sie in vielen Fällen eine plausible Annahme über die Datenverteilung darstellen (wieder aus obigem Grund).

A.9 Multivariate Gauß-Verteilung

(mehrdimensional)

Die eindimensionale Gaußverteilung einer Zufallsvariablen x mit Mittelwert μ und Varianz σ^2 nach (A.50) läßt sich durch die folgende Definition auf beliebig viele Dimensionen D verallgemeinern:

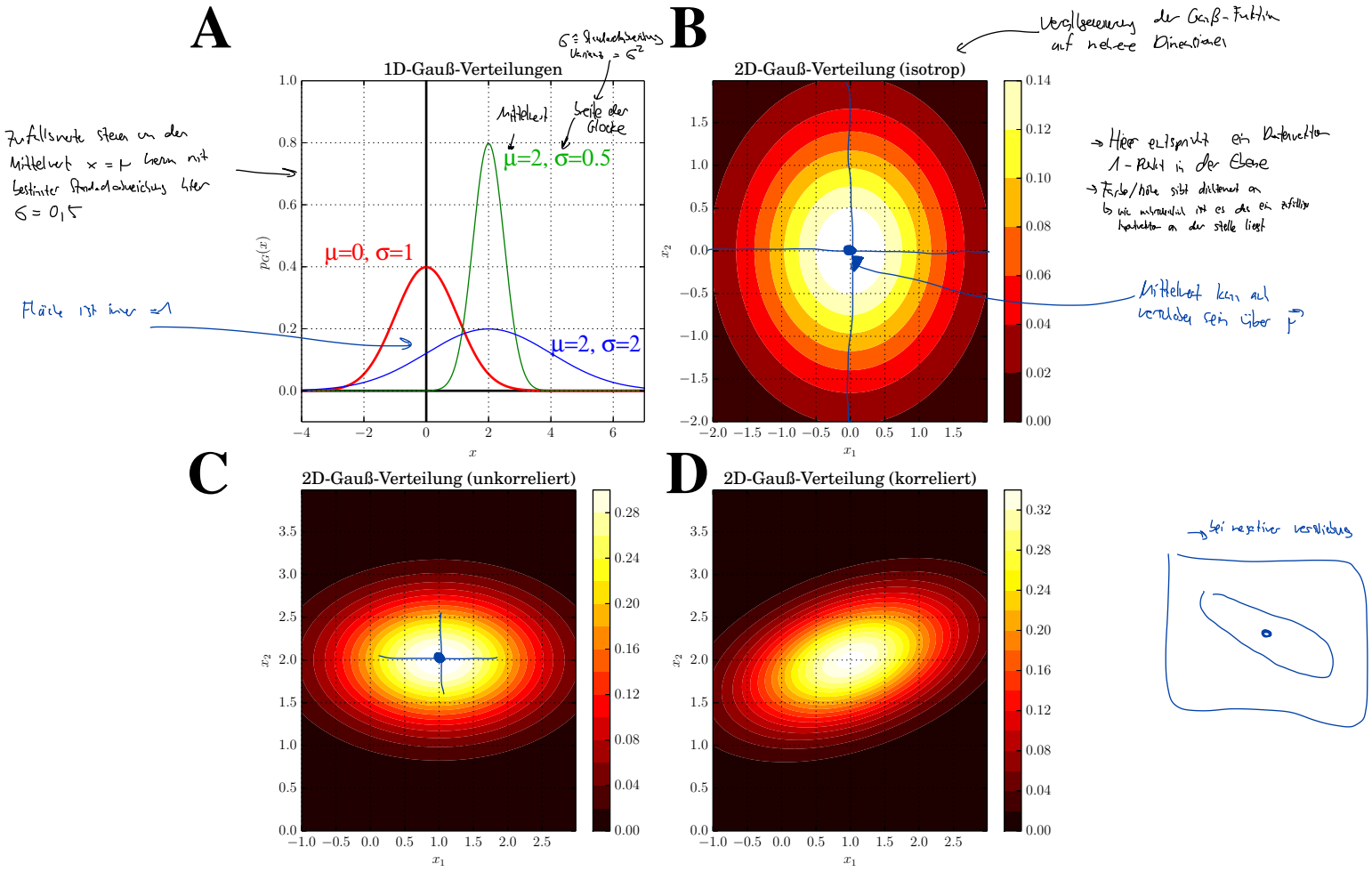
$$p_G(\mathbf{x}) := \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) := \frac{1}{\sqrt{(2\pi)^D |\boldsymbol{\Sigma}|}} \exp\left\{-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^{\mathrm{T}} \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})\right\} \qquad \text{(A.51)}$$

wobei der Zufallsvektor $\mathbf{x} = (x_1 \ \cdots \ x_D)$ aus D Zufallsvariablen x_i besteht, $\boldsymbol{\mu} = (\mu_1 \ \cdots \ \mu_D)$ deren Mittelwerten entspricht, $\boldsymbol{\Sigma}$ die symmetrische positiv semidefinite $D \times D$ Kovarianzmatrix ist (d.h. $(\boldsymbol{\Sigma})_{ii}$ ist die Varianz von x_i, und $(\boldsymbol{\Sigma})_{ij} = (\boldsymbol{\Sigma})_{ji}$ ist die Kovarianz von x_i und x_j) und $|\boldsymbol{\Sigma}|$ deren Determinante ist. Tatsächlich kann man mit etwas Linearer Algebra und Analysis leicht zeigen (Beweise siehe z.B. Bishop, 2006, Kap. 2.3, S. 78-84):

- p_G ist normiert und damit eine Dichtefunktion, d.h. $\int_{\mathbf{x} \in \mathbb{R}^D} p_G(\mathbf{x}) d\mathbf{x} = 1$.
- Für $D = 1$ ergibt (A.51) als Spezialfall genau die eindimensionale Gauß-Dichte (A.50), wobei $\boldsymbol{\mu} = (\mu)$ und $\boldsymbol{\Sigma} = (\sigma^2)$ bzw. $\boldsymbol{\Sigma}^{-1} = (\frac{1}{\sigma^2})$.
- Für beliebige $D \geq 1$ und $i, j \in \{1, \ldots, D\}$ gilt

$$E(x_i) = \mu_i \ , \qquad \mathrm{Var}(x_i) = \Sigma_{ii} \ , \qquad \mathrm{Cov}(x_i, x_j) = \Sigma_{ij}$$

d.h. der Mittelwertvektor $\boldsymbol{\mu}$ hat als Komponentenwerte die Mittelwerte der einzelnen Zufallsvariablen x_i, und die Kovarianzmatrix $\boldsymbol{\Sigma}$ enthält auf der Hauptdiagonalen die Varianzen der x_i und ansonsten die Kovarianzen von Zufallsvariablen-



Beispiele: Obige Skizze zeigt einige Gauß-Verteilungen.
(A) zeigt drei verschiedene eindimensionale Verteilungen nach (A.50) mit Erwartungswerten und Standardabweichungen $\mu = 0, \sigma = 1$ (rot; Standardnormalverteilung), $\mu = 2, \sigma = 0.5$ (grün) und $\mu = 2, \sigma = 2$ (blau).
(B) zeigt den Kontur-Plot einer zweidimensionale Verteilung nach (A.51) mit Erwartungen $\boldsymbol{\mu} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ und Kovarianzmatrix $\boldsymbol{\Sigma} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$. Solche Verteilungen nennt man auch isotrope Verteilungen da die Dichte in alle Raumrichtungen gleichartig abfällt (Konturlinien bilden *Kreise* um den Erwartungsvektor).
(C) zeigt eine zweidimensionale Gauß-Verteilung mit $\boldsymbol{\mu} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$ und $\boldsymbol{\Sigma} = \begin{pmatrix} 1 & 0 \\ 0 & 0.25 \end{pmatrix}$. Da die Varianz in x_1-Richtung größer als in x_2-Richtung ist, fällt auch die Dichte in

x_1-Richtung langsamer ab als in x_2-Richtung. Entsprechend sind die Konturlinien nun *Ellipsen*. Da die Kovarianzmatrix eine Diagonalmatrix mit $\mathrm{Cov}(x_1, x_2) = 0$ ist, nennt man so eine Verteilung unkorreliert, wobei hier die Zufallsvariablen x_1 und x_2 sogar unabhängig sind [5] Entsprechend verlaufen die Hauptachsen der Ellipsen in Richtung der Koordinatenachsen x_1 bzw. x_2.

(D) zeigt eine allgemeine Gauß-Verteilung mit $\boldsymbol{\mu} = \begin{pmatrix}1\\2\end{pmatrix}$ und $\boldsymbol{\Sigma} = \begin{pmatrix}1 & 0.2\\0.2 & 0.25\end{pmatrix}$ mit positiv korrelierten Komponenten x_1 und x_2 (d.h. wenn x_1 groß ist, dann ist x_2 mit hoher Wahrscheinlichkeit auch groß). Hier sind die Konturlinien *beliebige Ellipsen* und x_1 und x_2 sind voneinander abhängig.

A.10 Fehlerabschätzung und Error Function

Oft nimmt man bei statistischen Tests als Hypothese an, dass eine Messgröße X eine bestimmte Verteilung mit Dichte $p(x)$ hat. Um diese Hypothese zu widerlegen kann man ausrechnen mit welcher Wahrscheinlichkeit

$$P_{\mathrm{conf}} := P(X \in (x_{\min}; x_{\max}) = \int_{x_{\min}}^{x_{\max}} p(x)\,dx \tag{A.52}$$

die Messung X in einem gegebenen Konfidenzintervall $(x_{\min}; x_{\max})$ liegt. Werden $x_{\min}, x_{\max}$ so gewählt, dass P_{conf} groß ist (≈ 1) aber X trotzdem außerhalb des Konfidenzintervalls liegt, dann kann die Hypothese mit großer Sicherheit als widerlegt gelten. Genauer: Entscheidet man sich die Hypothese abzulehen, so ist die entsprechende Fehler-Wahrscheinlichkeit (einer falschen Entscheidung) das Komplement $P^C_{\mathrm{conf}} := 1 - P_{\mathrm{conf}}$.

Aufgrund des Zentralen Grenzwertsatzes ist in vielen Fällen die Hypothese X sei **Gauß-verteilt** mit Mittelwert μ und Varianz σ^2. Mit der Dichte $p(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$ nach (A.50) und mit der Substitution $y := \frac{x-\mu}{\sqrt{2}\sigma}$ und $\frac{dy}{dx} = \frac{1}{\sqrt{2}\sigma}$ ist dann

$$\begin{aligned} P_{\mathrm{conf}}(v) := P(X \in [\mu - v\sigma; \mu + v\sigma]) &= \int_{\mu - v\sigma}^{\mu + v\sigma} p(x)\,dx \\ &= \frac{1}{\sqrt{2\pi}} \int_{-v}^{v} e^{-y^2}\,dy = \frac{2}{\sqrt{\pi}} \int_0^v e^{-y^2}\,dy =: \mathrm{erf}(v) \end{aligned} \tag{A.53}$$

die Wahrscheinlichkeit, dass der Messwert X höchstens v Standardabweichungen σ vom Mittelwert μ entfernt ist, wobei $\mathrm{erf}(v)$ die sogenannte Error Function oder Fehlerfunktion ist (nicht zu verwechseln mit den Fehler- oder Verlustfunktionen etwa

[5] Denn man kann leicht nachprüfen, dass die Dichte $p_G(x_1, x_2) = \mathcal{N}(x_1; 1, 1) \cdot \mathcal{N}(x_2; 2, 0.5)$ als Produkt der beiden Randverteilungen von x_1 und x_2 geschrieben werden kann (vgl.(A.12) auf Seite 217; vgl. (A.27) auf Seite 219).

von Neuronalen Netzen!). Für $\mu + v\sigma = X$ oder $v = \frac{X-\mu}{\sigma}$ ist dann die komplementäre ("Irrtums-")Wahrscheinlichkeit [6]

$$P^C_{\text{conf}}(v) := 1 - P_{\text{conf}}(v) = \frac{2}{\sqrt{\pi}} \int_v^\infty e^{-y^2}\, dy = 1 - \text{erf}(v) =: \text{erfc}(v) \tag{A.54}$$

bei Ablehnung der Hypothese. Die unten folgende Skizze zeigt $\text{erf}(v)$ sowie die Komplementäre Fehlerfunktion $\text{erfc}(v)$.

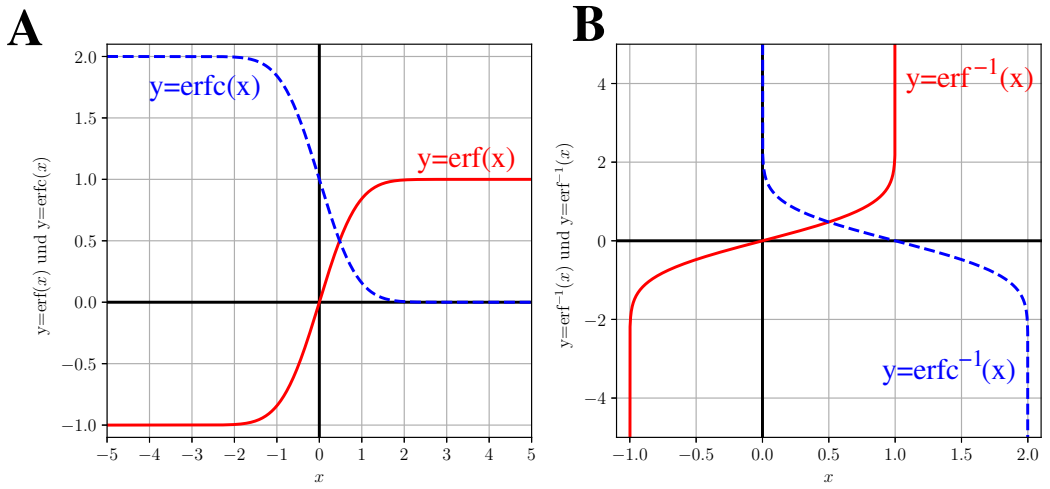
Beispiele: a) Eine Theorie sagt voraus, dass ein Messergebnis X Gauß-verteilt mit Mittelwert $\mu = 20$ und Varianz $\sigma^2 = 5$ sei. Wenn nun die Messung $X = 35$ ergibt und man deshalb die Theorie ablehnt, mit welcher Wahrscheinlichkeit irrt man sich dann? b) Angenommen die genannte Theorie würde stimmen: Bestimmen Sie d sodass X mit Wahrscheinlichkeit $P_{\text{conf}} = 0.9$ im Intervall $[\mu - d; \mu + d]$ liegt. Wie wäre d für $P_{\text{conf}} = 0.99$?

Lösung: a) $v = \frac{X-\mu}{\sigma} = \frac{45-35}{\sqrt{5}} \approx 4.472$ und nach (A.54) damit $P^C_{\text{conf}} = \text{erfc}(4.472) = 2.54e-10 \approx 0$, d.h. man kann sich (fast) sicher sein, dass die Theorie falsch ist.

b) Aus (A.53) folgt mit $d = v\sigma$ die Gleichung $\text{erf}(v) \stackrel{!}{=} P_{\text{conf}}$. Auflösen nach v ergibt

$$v = \text{erf}^{-1}(P_{\text{conf}}) \qquad \text{und damit} \quad d = v\sigma = \sigma\, \text{erf}^{-1}(P_{\text{conf}})\ . \tag{A.55}$$

Für $P_{\text{conf}} = 0.9$ gilt also $d = \sqrt{5}\,\text{erf}^{-1}(0.9) \approx 2.60$, d.h. X liegt mit Wahrscheinlichkeit 0.9 im Intervall $[20 - 2.60; 20 + 2.60]$ bzw. $[17.40; 37.40]$. Für $P_{\text{conf}} = 0.99$ ergibt sich entsprechend $d = \sqrt{5}\,\text{erf}^{-1}(0.99) \approx 4.07$, d.h. X liegt mit Wahrscheinlichkeit 0.99 im Intervall $[15.93; 35.93]$. Wie erf und erfc sind auch die inversen Funktionen erf^{-1} und erfc^{-1} Standardfunktionen die in vielen numerischen Bibliotheken (wie Matlab oder Numpy) zur Verfügung stehen (siehe folgende Skizze).



[6] Unter der Annahme dass X Gauß-verteilt mit Mittelwert μ und Varianz σ^2 sei, ist $v = \frac{X-\mu}{\sigma}$ nach (A.3,A.7) offenbar standard-normalverteilt, d.h. Gauß-verteilt mit Erwartungswert 0 und Varianz 1. Siehe auch den Beweis zu (A.50) auf Seite 225.

Anhang B

Mehrdimensionale Differentialrechnung

B.1 Ableitungen von Funktionen $f : \mathbb{R}^n \to \mathbb{R}^m$

Wir fassen kurz die wichtigsten Resultate vom Skript Skript Mathe-I, ab S. 219, Kapitel 9.1-9.3 über mehrdimensionale Differentialrechnung zusammen:

I) Die Ableitung $Df(\mathbf{x})$ einer Funktion $f : \mathbb{R}^n \to \mathbb{R}^m$ im Punkt $\mathbf{x}$ wird definiert als die $m \times n$ Matrix $\mathbf{J_f}(\mathbf{x})$ der linearen "Tangential"-Abbildung welche f in $\mathbf{x}$ approximiert. Man nennt $\mathbf{J_f}$ auch die Jacobi-Matrix von f.

II) Diese Definition verallgemeinert offenbar den eindimensionalen Fall $f(x) : \mathbb{R} \to \mathbb{R}$ mit $m = n = 1$, für den die Ableitung $\mathbf{J_f}(x) = f'(x)$ einfach als die Tangentensteigung in x definiert wird (siehe Skript Mathe-I, S. 112, Def. 5.1).

III) Für $f : \mathbb{R}^n \to \mathbb{R}$ ist $\mathbf{J_f}$ ein Zeilenvektor der Länge n, welchen man auch als Gradient

$$\text{grad } f(\mathbf{x}) := \nabla f(\mathbf{x}) := (f_{x_1}(\mathbf{x}) \cdots f_{x_n}(\mathbf{x})) \ . \tag{B.1}$$

bezeichnet. Hierbei ist $f_{x_i}(\mathbf{x}) := \frac{\partial f}{\partial x_i}(\mathbf{x})$ die partielle Ableitung von f in Richtung der i-ten Koordinatenachse, d.h. die Ableitung der eindimensionalen Funktion $f(x_i)$ bei der alle Variablen außer x_i als Konstanten aufgefasst werden.

IV) Der Gradient $\nabla f(\mathbf{x})$ entspricht der Richtung des steilsten Anstiegs:

- Um die Funktion $f(\mathbf{x})$ zu maximieren bewegt man sich von einem Punkt $\mathbf{x}$ ausgehend in Richtung des Gradienten (Gradientenanstiegsverfahren).
- Um $f(\mathbf{x})$ zu minimieren geht man entsprechend entgegen der Richtung des Gradienten (Gradientenabstiegsverfahren).

- Senkrecht zum Gradienten $\nabla f(\mathbf{x})$ verlaufen die Höhenlinien, entlang denen die Funktion $f(\mathbf{x})$ konstant ist.
- In einem lokalen Maximum bzw. Minimum $\mathbf{x}$ gilt notwendig $\nabla f(\mathbf{x}) = \mathbf{0}$.

V) Für mehrdimensionale Funktionen $f : \mathbb{R}^n \to \mathbb{R}^m$ ist die Ableitung Df die $m \times n$ Jacobi-Matrix $\mathbf{J_f}$, deren i-te Zeile dem Gradient der i-ten Funktionskomponente $f_i : \mathbb{R}^n \to \mathbb{R}$ von f entspricht.
Es gelten dieselben Ableitungsregeln wie für eindimensionale Funktionen:

$$\text{I) Linearität: } D(c \cdot f + d \cdot g) = c \cdot (Df) + d \cdot (Dg) \tag{B.2}$$

$$\text{II) Leibniz-/Produktregel: } D(f \cdot g) = (Df) \cdot g + f \cdot (Dg) \tag{B.3}$$

$$\text{III) Quotientenregel: } D\frac{f}{g} = \frac{(Df) \cdot g - f \cdot (Dg)}{g^2} \tag{B.4}$$

$$\text{IV) Kettenregel: } (D(f \circ g))(\mathbf{x}) = (Df)(g(\mathbf{x})) \cdot (Dg)(\mathbf{x}) \tag{B.5}$$

VI) Für Funktionen $f : \mathbb{R}^n \to \mathbb{R}$ ist die zweite Ableitung (d.h. die Ableitung des Gradienten ∇) die sogenannte Hesse-Matrix

$$\mathbf{H_f} := D^2 f = \mathbf{J}_{\nabla \mathbf{f}} = \begin{pmatrix} f_{x_1 x_1}(\mathbf{x}) & f_{x_1 x_2}(\mathbf{x}) & \cdots & f_{x_1 x_n}(\mathbf{x}) \\ f_{x_2 x_1}(\mathbf{x}) & f_{x_2 x_2}(\mathbf{x}) & \cdots & f_{x_2 x_n}(\mathbf{x}) \\ \vdots & \vdots & \ddots & \vdots \\ f_{x_n x_1}(\mathbf{x}) & f_{x_n x_2}(\mathbf{x}) & \cdots & f_{x_n x_n}(\mathbf{x}) \end{pmatrix} \tag{B.6}$$

Die Hesse-Matrix ist symmetrisch, d.h. $f_{x_i x_j}(\mathbf{x}) = f_{x_j x_i}(\mathbf{x})$, sodass die Reihenfolge bei mehrfachen partiellen Ableitungen beliebig vertauschbar ist.

VII) Der Satz von Taylor (siehe Skript Mathe-I, S. 121, Def. 5.9 mit Satz 5.10) gilt auch für Funktionen $f : \mathbb{R}^n \to \mathbb{R}$. Insbesondere gibt es für jedes Punktepaar $\mathbf{x}$ und $\mathbf{x} + \boldsymbol{\Delta} x$ auf deren Verbindungslinie einen Punkt $\boldsymbol{\xi}$ sodass gilt:

- Mittelwertsatz (Taylor-Entwicklung 1.Ordnung):

$$f(\mathbf{x} + \boldsymbol{\Delta} x) = f(\mathbf{x}) + (Df)(\boldsymbol{\xi}) \cdot \boldsymbol{\Delta} x \tag{B.7}$$

- Taylor-Entwicklung 2.Ordnung:

$$f(\mathbf{x} + \boldsymbol{\Delta} x) = f(\mathbf{x}) + (Df)(\mathbf{x}) \cdot \boldsymbol{\Delta} x + \frac{1}{2} \boldsymbol{\Delta} x^{\mathrm{T}} \cdot (D^2 f)(\xi) \cdot \boldsymbol{\Delta} x \tag{B.8}$$

B.2 Optimierung von Funktionen $f : \mathbb{R}^n \to \mathbb{R}$

Wir fassen kurz die wichtigsten Resultate vom Skript Skript Mathe-I, ab S. 227, Kapitel 9.3-9.4 zur Optimierung mehrdimensionaler Funktionen zusammen:

I) Aus dem Satz von Taylor folgen Bedingungen für lokale Extrempunkte $\mathbf{x}$ einer Funktion $f : \mathbb{R}^n \to \mathbb{R}$:

- Notwendige Bedingung für lokales Extremum: $Df(\mathbf{x}) = \nabla f(\mathbf{x}) \stackrel{!}{=} \mathbf{0}$.
- Hinreichende Bedingung für lokales Extremum: Falls $\nabla f(\mathbf{x}) = \mathbf{0}$ und zusätzlich die Hesse-Matrix $\mathbf{H_f}(\mathbf{x}) := (D^2 f)(\mathbf{x})$ positiv definit bzw. negativ definit ist, d.h. für beliebiges $\mathbf{\Delta} x \in \mathbb{R}^n$

$$\mathbf{\Delta} x^{\mathrm{T}} \mathbf{H_f}(\mathbf{x}) \mathbf{\Delta} x \; > \; 0 \qquad \text{bzw.} \qquad \mathbf{\Delta} x^{\mathrm{T}} \mathbf{H_f}(\mathbf{x}) \mathbf{\Delta} x \; < \; 0$$

  gilt, so befindet sich bei $\mathbf{x}$ ein Minimum bzw. Maximum.
- Man beachte, dass obige Kriterien die bekannten Bedingungen $f'(x) = 0$ und $f''(x) > 0$ bzw $f''(x) < 0$ von eindimensionalen Funktione $f : \mathbb{R} \to \mathbb{R}$ verallgemeinern.
- Zum Test auf Definitheit: Eine Matrix $\mathbf{A}$ ist positiv bzw. negativ definit genau dann, wenn...
  - ... bei nach Gauß-Elimination eines Gleichungssystems mit Matrix $\mathbf{A}$ (ohne Zeilentausch) die Diagonalelemente alle positiv bzw. negativ sind.
  - ... alle ihre Eigenwerte $\lambda_1, \lambda_2, \ldots$ positiv bzw. negativ sind.

II) Optimierung mit Nebenbedingungen durch Lagrange-Multiplikatoren und das Theorem von Karush, Kuhn und Tucker (KKT-Theorem): Gegeben seien Funktionen $f : \mathbb{R}^n \to \mathbb{R}$, $g : \mathbb{R}^n \to \mathbb{R}^m$ und $h : \mathbb{R}^n \to \mathbb{R}^k$. Falls $f(\mathbf{x})$ in $\mathbf{x} \in \mathbb{R}^n$ ein lokales Maximum unter den m Nebenbedingungs-Gleichungen $g(\mathbf{x}) = \mathbf{0}$ und den k Nebenbedingungs-Ungleichungen $h(\mathbf{x}) \geq \mathbf{0}$ hat, so existieren Lagrange-Multiplikatoren $\lambda_1, \ldots, \lambda_m \in \mathbb{R}$ und $\mu_1, \ldots, \mu_k \in \mathbb{R}$ mit

$$\mu_i \geq 0 \qquad \text{und} \qquad \mu_i h_i(\mathbf{x}) = 0 \text{ für } i = 1, \ldots, k \tag{B.9}$$

sodass der Gradient der Lagrange-Funktion

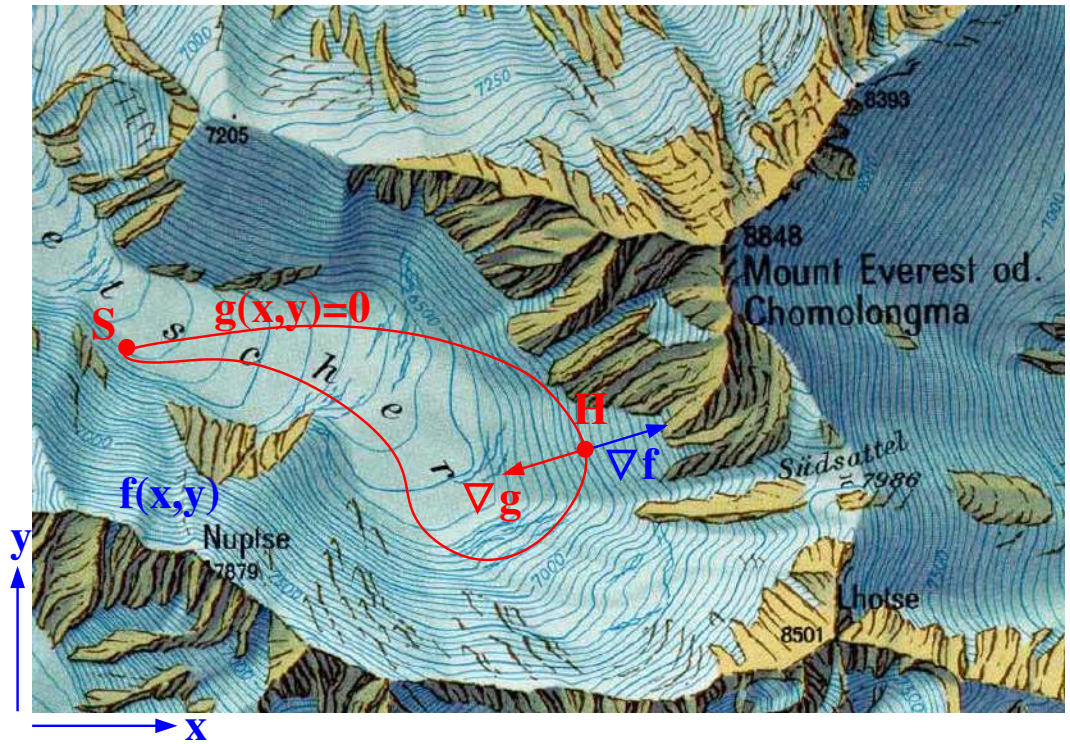
$$L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) := f(\mathbf{x}) + \boldsymbol{\lambda}^T g(\mathbf{x}) + \boldsymbol{\mu}^T h(\mathbf{x}) \tag{B.10}$$

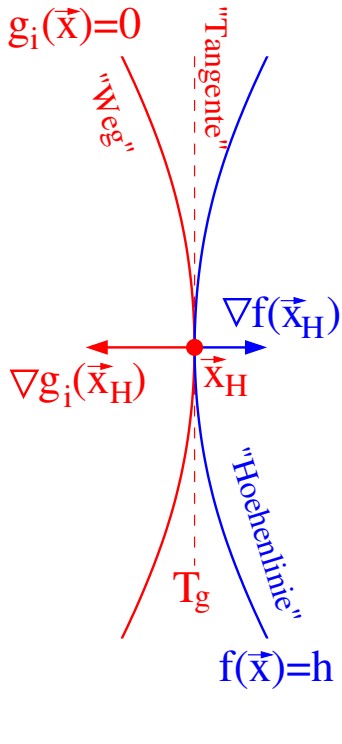
verschwindet, $\nabla L = \mathbf{0}$, bzw. äquivalent gilt:

$$\nabla f(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i \nabla g_i(\mathbf{x}) + \sum_{j=1}^{k} \mu_j \nabla h_j(\mathbf{x}) = \mathbf{0} \tag{B.11}$$

Bei Minimierungsproblemen mit Nebenbedingungen darf man einfach $f((\mathbf{x})$ durch $-f((\mathbf{x})$ ersetzen (dann Minima von f entsprechen Maxima von $-f$). Nach beidseitigem Multiplizieren der KKT-Bedingung (B.11) mit -1 ist dann klar, dass man für Minimierung auch einfach in (B.11) bzw. (B.10) die + durch − ersetzen darf.

III) Umformen von (B.11) ergibt insbesondere, dass für ein Optimum $\mathbf{x}$ sich der Gradient $\nabla f(\mathbf{x})$ der Zielfunktion immer als Linearkombinationen der Gradienten $\nabla g_i(\mathbf{x})$ bzw. $\nabla h_j(\mathbf{x})$ der Nebenbedingungsfunktionen darstellen läßt (d.h. von diesen linear abhängt; vgl. Skript Mathe-I,S. 184, Def. 8.12 von Linearer Unabhängigkeit). Insbesondere bedeutet dies für den Fall, dass nur eine einzige Nebenbedingung $g(\mathbf{x}) = 0 \in \mathbb{R}$ vorliegt (oder $h(\mathbf{x}) \geq 0 \in \mathbb{R}$), dass dann $\nabla f(\mathbf{x}) = -\lambda \nabla g(\mathbf{x})$ gilt (oder $\nabla f(\mathbf{x}) = -\mu \nabla h(\mathbf{x})$). D.h. in diesem Fall zeigen Gradienten von Zielfunktion f und Nebenbedingugsfunktion g (oder h) in entgegengesetzte Richtungen (siehe Skizze), und damit müssen die Höhenlinien bzw. Konturen von f und g_1 (oder h_1) in einem Optimum immer parallel verlaufen bzw. sich dort berühren (siehe folgende Skizze für 2-dim. Funktionen $f(x, y)$ und $g(x, y)$).





B.3 Vektor- und Matrixableitungen

Aufbauend auf den Resultaten vom Skript Mathe-I (vgl. vorige Abschnitte B.1 und B.2) wollen wir im folgenden Ableitungsregeln für Vektor- und Matrixfunktionen herleiten. Damit lassen sich sehr bequem etwa Bedingungen (d.h. Gleichungssysteme) für Extrema vektorwertiger Funktionen bestimmen. Die Ableitungen einer Vektorfunktion $\mathbf{a}$ oder Matrixfunktion $\mathbf{A}$ bezüglich eines *Skalars* x definiert man (in leichter Verallgemeinerung der Definitionen I,III,V in Abschnitt B.1) als

$$\left(\frac{\partial \mathbf{a}}{\partial x}\right)_i := \frac{\partial a_i}{\partial x} \qquad \text{und} \qquad \left(\frac{\partial \mathbf{A}}{\partial x}\right)_{ij} := \frac{\partial A_{ij}}{\partial x} \ . \tag{B.12}$$

Ganz ähnlich kann man Ableitungen bezüglich Vektoren und Matrizen definieren, z.B.

$$\left(\frac{\partial x}{\partial \mathbf{a}}\right)_i := \frac{\partial x}{\partial a_i} \qquad \text{und} \qquad \left(\frac{\partial x}{\partial \mathbf{A}}\right)_{ij} := \frac{\partial x}{\partial A_{ji}} \tag{B.13}$$

$$\left(\frac{\partial \mathbf{a}}{\partial \mathbf{b}}\right)_{ij} := \frac{\partial a_i}{\partial b_j} \qquad \text{und} \qquad \left(\frac{\partial \mathbf{a}}{\partial \mathbf{B}}\right)_{ijk} := \frac{\partial a_i}{\partial B_{kj}} \tag{B.14}$$

wobei die linken Seiten einfach den Definitionen von Gradient und Jacobi-Matrix entsprechen (siehe Abschnitt B.1,I,III,V). Damit lässt sich durch Ausschreiben obiger Definitionen leicht zeigen, dass für die Ableitung von Linearformen $\mathbf{a}^{\mathrm{T}}\mathbf{x} = \sum_i a_i x_i$

$$\frac{\partial}{\partial \mathbf{x}}(\mathbf{a}^{\mathrm{T}}\mathbf{x}) = \frac{\partial}{\partial \mathbf{x}}(\mathbf{x}^{\mathrm{T}}\mathbf{a}) = \mathbf{a}^{\mathrm{T}} \qquad \text{bzw.} \tag{B.15}$$

$$\frac{\partial}{\partial \mathbf{x}}(\mathbf{A}\mathbf{x}) = \mathbf{A} \tag{B.16}$$

gilt. **Beweis:** Für $\mathbf{x} = \begin{pmatrix} x_1 & \dots & x_n \end{pmatrix}^{\mathrm{T}}$ hat $f(\mathbf{x}) := \mathbf{a}^{\mathrm{T}}\mathbf{x} = \mathbf{x}^{\mathrm{T}}\mathbf{a} = \sum_{i=1}^{n} a_i x_i$ die partiellen Ableitungen $\frac{\partial f}{\partial x_i} = a_i$ und damit den Gradient $Df = \nabla f = \begin{pmatrix} a_1 & \cdots & a_n \end{pmatrix} = \mathbf{a}^{\mathrm{T}}$, womit (B.15) gezeigt ist. Damit folgt auch (B.16) da die Jacobi-Matrix von $\mathbf{A}\mathbf{x}$ in jeder Zeile $i = 1, 2, ...n$ den Gradienten $\nabla \mathbf{A}_i\mathbf{x} = \mathbf{A}_i$ stehen hat (siehe Abschnitt B.1,V). □
Ähnlich gilt für die Ableitung einer quadratischen Form $\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x} = \sum_{i,j} A_{ij} x_i x_j$

$$\frac{\partial}{\partial \mathbf{x}}(\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x}) = \mathbf{x}^{\mathrm{T}}(\mathbf{A} + \mathbf{A}^{\mathrm{T}}) \qquad \text{allgemein, und deshalb} \tag{B.17}$$

$$\frac{\partial}{\partial \mathbf{x}}(\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x}) = 2\mathbf{x}^{\mathrm{T}}\mathbf{A} \qquad \text{, falls } \mathbf{A} \text{ symmetrisch ist.} \tag{B.18}$$

$$\frac{\partial}{\partial \mathbf{x}}(\mathbf{x}^{\mathrm{T}}\mathbf{x}) = 2\mathbf{x}^{\mathrm{T}} \tag{B.19}$$

Beweis: (B.17) folgt aus $\frac{\partial}{\partial x_k}(\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x}) = \frac{\partial}{\partial x_k}\sum_{ij} A_{ij} x_i x_j = 2A_{kk}x_k + \sum_{i\neq k} A_{ik}x_i + \sum_{j\neq k} A_{kj}x_j = \sum_i A_{ik}x_i + \sum_j A_{kj}x_j = (\mathbf{A}^{\mathrm{T}}\mathbf{x})_k + (\mathbf{A}\mathbf{x})_k = (\mathbf{x}^{\mathrm{T}}\mathbf{A} + \mathbf{x}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}})_k$. Falls $\mathbf{A}$

symmetrisch ist folgt (B.18) wegen $\mathbf{A}^{\mathrm{T}} = \mathbf{A}$ aus (B.17). Damit folgt auch (B.19) für die (symmetrische) Einheitsmatrix $\mathbf{A} = \mathbf{I}$. □

Nach Ausmultiplizieren $(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\mathbf{A}(\mathbf{x}-\boldsymbol{\mu}) = \mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x} - \boldsymbol{\mu}^{\mathrm{T}}\mathbf{A}\mathbf{x} - \mathbf{x}^{\mathrm{T}}\mathbf{A}\boldsymbol{\mu} + \boldsymbol{\mu}^{\mathrm{T}}\mathbf{A}\boldsymbol{\mu}$ und entsprechendem Ableiten mit obigen Regeln, $\frac{\partial}{\partial \mathbf{x}}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\mathbf{A}(\mathbf{x}-\boldsymbol{\mu}) = \mathbf{x}^{\mathrm{T}}(\mathbf{A}+\mathbf{A}^{\mathrm{T}}) - \boldsymbol{\mu}^{\mathrm{T}}\mathbf{A} - (\mathbf{A}\boldsymbol{\mu})^{\mathrm{T}} = \mathbf{x}^{\mathrm{T}}(\mathbf{A}+\mathbf{A}^{\mathrm{T}}) - \boldsymbol{\mu}^{\mathrm{T}}(\mathbf{A}+\mathbf{A}^{\mathrm{T}})$ sieht man auch sofort [1]

$$\frac{\partial}{\partial \mathbf{x}}[(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\mathbf{A}(\mathbf{x}-\boldsymbol{\mu})] = (\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}(\mathbf{A}+\mathbf{A}^{\mathrm{T}}) \quad \text{allgemein, und deshalb} \tag{B.20}$$

$$\frac{\partial}{\partial \mathbf{x}}[(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\mathbf{A}(\mathbf{x}-\boldsymbol{\mu})] = 2(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\mathbf{A} \quad \text{, falls } \mathbf{A} \text{ symmetrisch ist.} \tag{B.21}$$

Bemerkung: Diese Vektorableitungen verallgemeinern die bekannten Ableitungsregeln $(ax)' = a$ und $(ax^2)' = 2ax$ für eindimensionale lineare und quadratische Funktionen (siehe Skript Mathe-I, S. 115, Satz 5.4.II).
Ebenso verallgemeinert sich die Produktregel der Differentialrechnung (B.3) zur Produktregel für Matrix-Ableitungen:

$$\frac{\partial}{\partial x}(\mathbf{A}\mathbf{B}) = \frac{\partial \mathbf{A}}{\partial x}\mathbf{B} + \mathbf{A}\frac{\partial \mathbf{B}}{\partial x}\,, \tag{B.22}$$

Beweis: Für $\mathbf{A} = \begin{pmatrix} \mathbf{a}_1 \\ \vdots \\ \mathbf{a}_m \end{pmatrix}$ und $\mathbf{B} = \begin{pmatrix} \mathbf{b}_1 & \cdots & \mathbf{b}_n \end{pmatrix}$ mit entsprechenden Zeilenvektoren $\mathbf{a}_i$ und Spaltenvektoren $\mathbf{b}_j$ kann man die Ableitung in Zeile i und Spalte j wie folgt berechnen: $(\frac{\partial}{\partial x}\mathbf{A}\mathbf{B})_{ij} = \frac{\partial}{\partial x}(\mathbf{a}_i\mathbf{b}_j) = (\frac{\partial}{\partial x}\mathbf{a}_i)\mathbf{b}_j + \mathbf{a}_i(\frac{\partial}{\partial x}\mathbf{b}_j) = (\frac{\partial \mathbf{A}}{\partial x})_i\mathbf{b}_j + \mathbf{a}_i(\frac{\partial \mathbf{B}}{\partial x})_j$. □

Durch wiederholtes Anwenden dieser Produktregel erhählt man für das Produkt aus N Matrizen $\mathbf{A}_i$ mit der abkürzenden Schreibweise $\mathbf{A}_i' := \frac{\partial}{\partial x}\mathbf{A}_i$ entsprechend

$$(\mathbf{A}_1 \cdots \mathbf{A}_N)' = \mathbf{A}_1'\mathbf{A}_2 \cdots \mathbf{A}_N + \mathbf{A}_1\mathbf{A}_2'\mathbf{A}_3 \cdots \mathbf{A}_N + \ldots + \mathbf{A}_1 \cdots \mathbf{A}_{N-1}\mathbf{A}_N'\,. \tag{B.23}$$

Beweis (durch Induktion über N): I.A.($N = 2$): Siehe (B.22). I.S. ($N \to N+1$): Es gilt $(\mathbf{A}_1 \cdots \mathbf{A}_{N+1})' \overset{(B.22)}{=} \mathbf{A}_1'(\mathbf{A}_2 \cdots \mathbf{A}_{N+1}) + \mathbf{A}_1(\mathbf{A}_2 \cdots \mathbf{A}_{N+1})' \overset{(I.V.)}{=} \mathbf{A}_1'\mathbf{A}_2 \cdots \mathbf{A}_N + \mathbf{A}_1(\mathbf{A}_2'\mathbf{A}_3 \cdots \mathbf{A}_{N+1} + \ldots \mathbf{A}_2 \cdots \mathbf{A}_N\mathbf{A}_{N+1}')$ was ausmultipliziert genau der rechten Seite von (B.23) entspricht. □

Beispiele zu den Ableitungsregeln: Sei jeweils $\mathbf{x} := \begin{pmatrix} x_1 & x_2 \end{pmatrix}^{\mathrm{T}}$.

a) Für $f(\mathbf{x}) = \begin{pmatrix} 1 & 2 \end{pmatrix}\mathbf{x} = x_1 + 2x_2$ ist $Df(\mathbf{x}) = \begin{pmatrix} 1 & 2 \end{pmatrix}$ nach (B.15).

b) Für $f(\mathbf{x}) = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}\mathbf{x} = \begin{pmatrix} x_1 + 2x_2 \\ 3x_1 + 4x_2 \end{pmatrix}$ ist $Df(\mathbf{x}) = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ nach (B.16).

[1] Dasselbe Ergebnis (B.20) erhält man auch unmittelbar aus (B.17) mit der Kettenregel (B.5) und der inneren Ableitung $\frac{\partial}{\partial \mathbf{x}}(\mathbf{x}-\boldsymbol{\mu}) = \frac{\partial}{\partial \mathbf{x}}\mathbf{x} = \frac{\partial}{\partial \mathbf{x}}\mathbf{I}\mathbf{x} = \mathbf{I}$ nach (B.16).

c) Für $f(\mathbf{x}) = \mathbf{x}^{\mathrm{T}} \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \mathbf{x} = 1 \cdot x_1 x_1 + 2 \cdot x_1 x_2 + 3 \cdot x_2 x_1 + 4 \cdot x_2 x_2 = x_1^2 + 5x_1 x_2 + 4x_2^2$
ist $Df(\mathbf{x}) = \mathbf{x}^{\mathrm{T}} \left(\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix} \right) = \begin{pmatrix} x_1 & x_2 \end{pmatrix} \begin{pmatrix} 2 & 5 \\ 5 & 8 \end{pmatrix} = \begin{pmatrix} 2x_1 + 5x_2 & 5x_1 + 8x_2 \end{pmatrix}$
nach (B.17).

d) Für $f(\mathbf{x}) = \mathbf{x}^{\mathrm{T}} \begin{pmatrix} 1 & 2 \\ 2 & 3 \end{pmatrix} \mathbf{x} = x_1^2 + (2+2)x_1 x_2 + 3x_2^2$
ist $Df(\mathbf{x}) = 2\mathbf{x}^{\mathrm{T}} \begin{pmatrix} 1 & 2 \\ 2 & 3 \end{pmatrix} = 2 \begin{pmatrix} x_1 + 2x_2 & 2x_1 + 3x_2 \end{pmatrix} = \begin{pmatrix} 2x_1 + 4x_2 & 4x_1 + 6x_2 \end{pmatrix}$
nach (B.18).

e) Für $f(\mathbf{x}) = \left(\mathbf{x} - \begin{pmatrix} 5 \\ 6 \end{pmatrix} \right)^{\mathrm{T}} \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \left(\mathbf{x} - \begin{pmatrix} 5 \\ 6 \end{pmatrix} \right) = (x_1 - 5)^2 + (2+3)(x_1 - 5)(x_2 - 6) + 4(x_2 - 6)^2$
ist $Df(\mathbf{x}) = \left(\mathbf{x} - \begin{pmatrix} 5 \\ 6 \end{pmatrix} \right)^{\mathrm{T}} \left(\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix} \right) = \left(\mathbf{x} - \begin{pmatrix} 5 \\ 6 \end{pmatrix} \right)^{\mathrm{T}} \begin{pmatrix} 2 & 5 \\ 5 & 8 \end{pmatrix} = \begin{pmatrix} 2(x_1 - 5) + 5(x_2 - 6) & 5(x_1 - 5) + 8(x_2 - 6) \end{pmatrix} = \begin{pmatrix} 2x_1 + 5x_2 - 40 & 5x_1 + 8x_2 - 73 \end{pmatrix}$
nach (B.20).

B.4 Die logistische Sigmoide und die Softmax-Funktion

Die logistische Sigmoide (3.6) von Seite 42 sowie die Softmax-Funktion (4.29) von Seite 96 spielen eine wichtige Rolle für probabilistische Klassifikationsmodelle, da durch sie die Klassenwahrscheinlichkeiten $p(\mathcal{C}_k|\mathbf{x})$ ausgedrückt werden können (siehe Kapitel **??**). Wir wollen in diesem Anhang die wichtigsten Eigenschaften dieser Funktionen zur späteren Referenz festhalten.

B.4.1 Die logistische Sigmoide

Logistische SigmoideSigmoidfunktion Als **logistische Sigmoide** bezeichnet man die Funktion $\sigma : \mathbb{R} \to (0; 1)$ mit

$$\sigma(a) := \frac{1}{1 + e^{-a}} \; . \tag{B.24}$$

Man bezeichnet sie auch als *Squashing-Funktion* (zu dt. Quetschungsfunktion), da sie die gesamten Achsen auf ein endliches Intervall $(0, 1)$ abbildet. Man erkennt dies, da alle Terme in (3.6) positiv sind, und der Nenner immer größer als der Zähler ist. Man sieht außerdem, dass $\sigma(a)$ monoton steigend in a ist, da der Nenner für zunehmendes a immer größer wird. Aus der Stetigkeit von $\sigma(a)$ folgt deswegen die Umkehrbarkeit, wobei man die sogenannte *Logit-Funktion* als **Umkehrfunktion** $\sigma^{-1}(y)$ durch Auflösen

der Gleichung $y = \frac{1}{1+e^{-a}}$ nach a erhält, [2]

$$\sigma^{-1}(y) := \ln\left(\frac{y}{1-y}\right) . \tag{B.25}$$

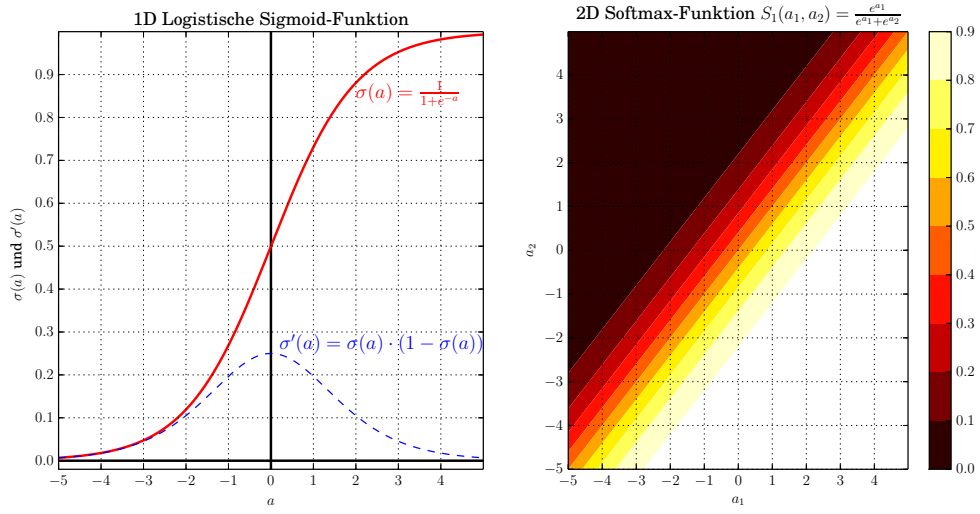
Weiter erfüllt die Sigmoidfunktion die **Symmetrie**

$$\sigma(-a) = 1 - \sigma(a) \tag{B.26}$$

denn $1 - \sigma(a) = 1 - \frac{1}{1+e^{-a}} = \frac{1+e^{-a}-1}{1+e^{-a}} = \frac{e^{-a}\cdot e^{a}}{(1+e^{-a})\cdot e^{a}} = \frac{1}{1+e^{a}} = \sigma(-a)$. Weiter hat die logistische Sigmoidfunktion die **Ableitung**

$$\frac{d\sigma(a)}{da} = \sigma(a)\cdot(1-\sigma(a)) \tag{B.27}$$

denn mit der Quotientenregel folgt $\sigma'(a) = \frac{(1)'\cdot(1+e^{-a})-1\cdot(1+e^{-a})'}{(1+e^{-a})^2} = \frac{e^{-a}}{(1+e^{-a})^2} = \sigma(a)\frac{e^{-a}}{1+e^{-a}} = \sigma(a)\frac{1}{1+e^{a}} = \sigma(a)\sigma(-a)$. Die folgende Skizze (linker Teil) zeigt $\sigma(a)$ und ihre Ableitung $\sigma'(a)$.



B.4.2 Die Softmax-Funktion

Softmax-Funktion Als Softmax-Funktion der K Aktivierungs-Potentiale $\mathbf{a} := \begin{pmatrix} a_1 & \cdots & a_K \end{pmatrix}^{\mathrm{T}}$ bezeichnet man die Funktion $\mathcal{S} : \mathbb{R}^K \to (0,1)^K$, welche für $k = 1, ..., K$

[2] Aus $\sigma : \mathbb{R} \to (0;1)$ und der Umkehrbarkeit folgt aus dem "Squashing" nebenbei, dass das Intervall $(0;1)$ genau gleich viele Zahlen enthält wie $\mathbb{R}$. Mit der skalierten und verschobenen Sigmoidfunktion $b\sigma(a) + c \in (c; c+b)$ folgt für $b > 0$ sogar, dass jedes Intervall $(c; c+b)$ gleich viele Zahlen wie $\mathbb{R}$ enthält, selbst wenn $b \approx 0$ sehr klein ist.

komponentenweise durch

$$\mathcal{S}_k(a_1, ..., a_K) := \frac{e^{a_k}}{\sum_j e^{a_j}} = \frac{1}{\sum_j \frac{e^{a_j}}{e^{a_k}}} = \frac{1}{\sum_j e^{-(a_k - a_j)}} \in (0, 1) \tag{B.28}$$

definiert ist (vgl. (4.29) auf Seite 96 und obige Skizze (rechter Teil). Man sieht leicht, dass die Softmax-Funktion geeignet ist eine **Wahrscheinlichkeitsverteilung** über K Zufallsvariablen darszustellen, denn offenbar gilt:

- $\mathcal{S}(\mathbf{a}) \in (0; 1)^K$, da jede Komponente $\mathcal{S}_k(\mathbf{a}) > 0$ der positiv ist (denn alle Terme $e^{a_k} > 0$ und $e^{a_j} > 0$ sind positiv) und außerdem auch kleiner als 1 bleiben (denn Nenner ist immer größer als der Zähler, da der Zähler-Term auch im Nenner vorkommt).
- Die Komponenten summieren zu 1, d.h. $\sum_{k=1}^{K} \mathcal{S}_k(\mathbf{a}) = 1$, da bei der Summe sich alle Zähler zur selben Summe wie im Nenner addieren.

Der Name Softmax kommt daher, dass $\mathcal{S}$ tatsächlich eine **weiche Maximum-Funktion** berechnet: Da die Exponentialfunktion e^a sehr steil mit a ansteigt, gilt allgemein $e^a + e^b \approx e^a$ falls a wesentlich größer als b ist. Falls also eine Input-Komponente a_k dominiert und wesentlich grøßer ist als alle anderen Komponenten a_j, dann gilt für den Nenner $\sum_j e^{a_j} \approx e^{a_k}$ und damit $\mathcal{S}_k(\mathbf{a}) \approx 1$ während für $j \neq k$ alle anderen Output-Komponenten $\mathcal{S}_j(\mathbf{a}) \approx 0$ sind. Nur falls keine Komponente dominiert und mehrere Inputs a_j dieselbe Größenordnung haben sind alle Output-Komponenten $\mathcal{S}_k(\mathbf{a}) \ll 1$ deutlich kleiner als 1. Da die Exponentialfunktion e^a streng monoton steigend ist, ändert sich jedoch niemals die **Ordnung** der Output-Komponenten gegenüber der Input-Komponenten,

$$a_k > a_j \qquad \Leftrightarrow \qquad \mathcal{S}_k(\mathbf{a}) > \mathcal{S}_j(\mathbf{a}) \tag{B.29}$$

Da die Softmax-Funktion vom Typ $\mathcal{S} : \mathbb{R}^K \to (0, 1)^K$ ist, stellt ihre **Ableitung** $D\mathcal{S}(\mathbf{a})$ eine volle $K \times K$ **Jacobi-Matrix** dar (siehe Anhang B.1), in deren i-ten Zeile und j-ten Spalte die partielle Ableitung

$$(D\mathcal{S})_{ij} = \frac{\partial \mathcal{S}_i}{\partial a_j} = \frac{\partial}{\partial a_j} \frac{e^{a_i}}{\sum_{j'} e^{a_{j'}}} \tag{B.30}$$

steht. Wir müssen hierbei zwei Fälle unterscheiden:
1) Für $i = j$ folgt mit der Quotientenregel

$$\begin{aligned}(D\mathcal{S})_{ii} = \frac{\partial \mathcal{S}_i}{\partial a_i} &= \frac{e^{a_i} \sum_{j'} e^{a_{j'}} - e^{a_i} e^{a_i}}{\left(\sum_{j'} e^{a_{j'}}\right)^2} \\ &= \frac{e^{a_i} \left(\sum_{j'} e^{a_{j'}} - e^{a_i}\right)}{\left(\sum_{j'} e^{a_{j'}}\right) \cdot \left(\sum_{j'} e^{a_{j'}}\right)} = \mathcal{S}_i(\mathbf{a}) \cdot (1 - \mathcal{S}_i(\mathbf{a}))\end{aligned}$$

2) Für $i \neq j$ folgt ähnlich

$$(D\mathcal{S})_{ij} = \frac{\partial \mathcal{S}_i}{\partial a_j} = \frac{0 - e^{a_i} e^{a_j}}{\left(\sum_{j'} e^{a_{j'}}\right)^2}$$
$$= \frac{-e^{a_i} e^{a_j}}{\left(\sum_{j'} e^{a_{j'}}\right) \cdot \left(\sum_{j'} e^{a_{j'}}\right)} = -\mathcal{S}_i(\mathbf{a}) \cdot \mathcal{S}_j(\mathbf{a})$$

Zusammen ergibt sich also die **Ableitungsformel** für die Softmaxfunkton

$$(D\mathcal{S})_{ij} = \mathcal{S}_i(\mathbf{a}) \cdot (I_{ij} - \mathcal{S}_j(\mathbf{a})) \ , \tag{B.31}$$

wobei I_{ij} eine Komponente der Einheitsmatrix $\mathbf{I}$ ist (also 1 falls $i = j$ und 0 sonst).

Zusammenhang zwischen Softmax-Funktion und logistischer Sigmoidfunktion Beachten Sie die Ähnlichkeit von (B.31) zur Ableitungsformel (B.27) der logistischen Sigmoidfunktion, die sich aus der engen Verwandtschaft zwischen $\sigma(a)$ und $\mathcal{S}(\mathbf{a})$ aufgrund der involvierten Expontialterme ergibt. Genauer gilt offenbar

$$\sigma(a) := \frac{1}{1 + e^{-a}} = \frac{e^a}{(1 + e^{-a}) \cdot e^a} = \frac{e^a}{e^a + 1} = \frac{e^a}{e^a + e^0} = \mathcal{S}_1(a, 0)$$

d.h. die logistische Sigmoidfunktion $\sigma(a)$ liefert denselben Wert wie die Softmax-Funktion für $K = 2$ Klassen angewandt auf die Diskriminanzwerte $\mathbf{a} = \begin{pmatrix} a & 0 \end{pmatrix}$.

B.4.3 Numerische Berechnung der Softmax-Funktion

Da $e^a \to \infty$ sehr schnell für $a \to \infty$ divergiert, würde eine naive Berechnung $\frac{e^{a_k}}{\sum_j e^{a_j}}$ nach (B.28) bei endlicher Mantissen-Breite sehr leicht zu Überläufen bzw. Werten $\frac{\infty}{\infty}$ =NaN ("not-a-number") führen falls ein oder mehrere der Potentiale a_j groß sind. Um dies zu vermeiden kann man den Bruch $\frac{e^{a_k}}{\sum_j e^{a_j}}$ mit dem Faktor e^{-c} erweitern, wobei $c := \max\{a_j | j = 1, ..., K\}$ das Maximum aller Potentiale a_j ist. Dann gilt

$$\mathcal{S}_k(a_1, ..., a_K) = \frac{e^{a_k - c}}{\sum_j e^{a_j - c}} \tag{B.32}$$

sodass alle Exponenten $a_j - c \leq 0$ sind und der größte Exponent den Wert 0 hat. Damit sind alle Exponentialterme ≤ 1 und einer hat genau den Wert 1. Entsprechend können keine Überläufe mehr auftreten. Unterläufe bei denen kleine Werte durch 0 angenähert werden lassen sich allerdings nicht vermeiden. In Python/Numpy kann die Softmax-Funktion wie folgt mit nur zwei Zeilen Code implementiert werden:

```
import numpy as np

def softmax(a):  # compute softmax function for potential vector a; for numerical stability
    e_a = np.exp(a - np.max(a))      # subtract maximum potential such that max. exponent is 1
    return e_a / e_a.sum()           # return softmax function value
```

B.4.4 Invertierung der Softmax-Funktion

Im Gegensatz zur logistischen Sigmoidfunktion kann man die Softmax-Funktion nicht einfach invertieren, denn nach (B.32) gilt $\mathcal{S}(\mathbf{a}) = \mathcal{S}(\mathbf{a} + C)$ für jede Konstante $C \in \mathbb{R}$. Die folgende Betrachtung zeigt, dass die Invertierung bis auf diese additive Konstante C eindeutig ist: Um die Notation schlank zu halten definieren wir wir $x_k := e^{a_k}$ und damit $y_k := \mathcal{S}_k(\mathbf{a}) = \frac{x_k}{x_1+\ldots+x_K}$ für $k = 1, \ldots, K$. Für die Invertierung nehmen wir an, dass alle y_k gegeben sind, und die x_k gesucht sind. Umformen der letzen Gleichung ergibt äquivalent $y_k(x_1 + \ldots x_K) - x_k = 0$ oder nach Teilen durch $y_k \neq 0$

$$(1 - \frac{1}{y_k})x_k + \sum_{l \neq k} x_l = 0 \qquad \text{für } k = 1, \ldots, K$$

Subtrahiert man von der k-ten Gleichung die i-te Gleichung bleibt nur $(1 - \frac{1}{y_k} - 1)x_k + (1 - (1 - \frac{1}{y_i}))x_i = 0$ oder

$$\frac{x_i}{y_i} - \frac{x_k}{y_k} = 0 \qquad \text{bzw.} \qquad x_k = \frac{y_k}{y_i} x_i \qquad \text{für } k, i = 1, \ldots, K$$

stehen. Daran erkennt man, dass man ein $x_i = e^{a_i} > 0$ bzw. $a_i := C' \in \mathbb{R}$ beliebig wählen kann, und daraus folgen dann alle anderen $x_k = e^{a_k} = \frac{y_k}{y_i} x_i$ bzw.

$$a_k = \ln(\frac{y_k}{y_i} x_i) = \ln(y_k) - \ln(y_i) + C' = \ln(y_k) + C \tag{B.33}$$

wobei wir $C' := C + \ln(y_i)$ für eine beliebige Konstante $C \in \mathbb{R}$ wählen dürfen. Interpretiert man die Outputs y_k der Softmax-Funktion als Wahrscheinlichkeiten, dann sind die a_k für $C := 0$ die Log-Probabilities. Und für $C := -\max_k \ln(y_k)$ ist $a_k = \ln \frac{y_k}{\max_k y_k}$ das Log-Odds-Ratio von y_k zur wahrscheinlichsten Klasse. Letzteres ist konsistent mit der numerisch stabilen Darstellung der Softmax-Funktion nach (B.32).

Stichwortverzeichnis