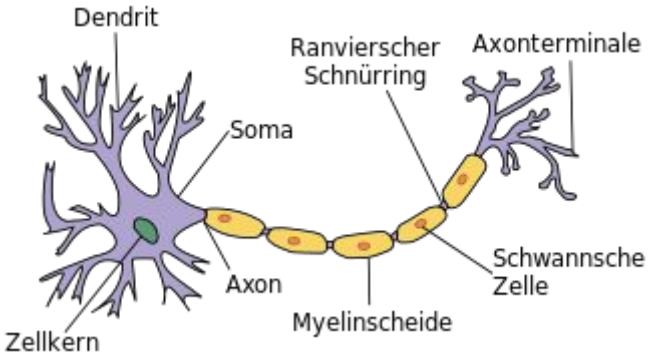
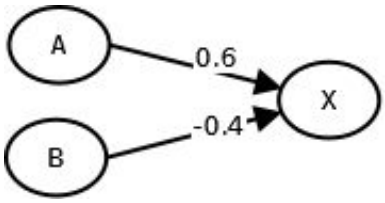
Biologische Grundlagen:



Ein **Neuron** empfängt mehrere Signale über seine Dendriten.
Nach der Verarbeitung der Signaleingänge sendet das Neuron ein Signal über das (nur einfach vorhandene) **Axon**.
Der Informationsübergang findet an der **Synapse** statt: dieser Übergang kann das Signal erregen oder hemmen.

Modellierung im Computer



Beispiel:
Wir haben die drei Neuronen A, B und X, wobei A und B jeweils ihr Signal an X weiterleiten.
Die Synapsen haben die **Gewichte** 0.6 und -0.4.
Wenn zum Zeitpunkt t=0 die Zelle A nun einen **Aktivierungszustand** von 0.5 besitzt und Zelle B einen Aktivierungzustand von 0.2 besitzt, so erhält die Zelle X folgenden Input:
$0.6 \cdot 0.5 + (-0.4) \cdot 0.2 = 0.22$
Dies erinnert an ein Skalaprodukt (oder allgemeiner: an ein Matrix*Vektor-Produkt), und tatsächlich könnte man die Rechnung auch folgendermaßen schreiben:

$$\begin{pmatrix} 0{,}6 & -0{,}4 \end{pmatrix} \cdot \begin{pmatrix} 0{,}5 \\ 0{,}2 \end{pmatrix} = 0.22$$

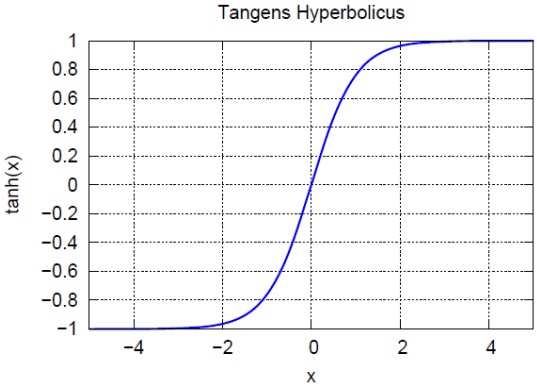
(Dabei ist der erste Faktor als Matrix mit einer Zeile aber zwei Spalten zu verstehen)
Diese Rechenart wird sich später als nützlich erweisen.

Die Entscheidung, dass alle Ausgänge (multipliziert mit den Gewichten) einfach in der Zelle X addiert werden ist eine Möglichkeit unter vielen (warum nicht multiplizieren?). Die Funktion, die die n Eingabewerte (im Beispiel: n=2) auf eine reelle Zahl abbildet (also eine Funktion, welche die Zahlenmengen $R^n \rightarrow R$ abbildet) heißt **Propagierungsfunktion**.

Auf das Ergebnis der Propagierungsfunktion wird nun die **Aktivierungsfunktion** angewendet um den neuen Aktivierungszustand zu berechnen. Da das Ergebnis der Propagierungsfunktion ja beliebig groß werden kann (z.B. wenn man nicht nur zwei Neuronen A und B hat, die auf X einwirken, sondern mehrere tausend) bewirkt die Aktivierungsfunktion oft eine "Normierung" des Signals auf einen bestimmten Wertebereich. Die einfachste Aktivierungsfunktion ist die sogenannte **Schwellenwertfunktion**. Man wählt einen Schwellwert, z.B. 0.15 und legt fest:

$$f(x) = \begin{cases} 0 & \text{für } x < 0{,}15 \\ 1 & \text{für } x \geq 0{,}15 \end{cases}$$

In unserem Beispiel würde für das Ergebnis 0.22 aus unserer Propagierungsfunktion der Wert 1 erzeugt, da $0.22 > 0{,}15$ ist.
Damit erhält man nur noch die Aktivierungszustände 0 und 1.



In der Praxis benötigt man für bestimmte Lernverfahren, mit dem nan das künstliche neuronale Netz trainieren will, aber **differenzierbare** Aktivierungsfunktionen.
Beliebt ist z.B. der Tangens Hyperbolicus.

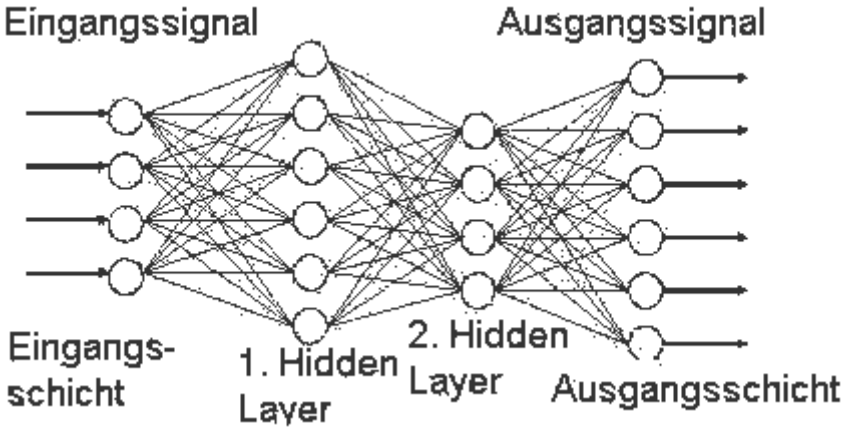
Man kann schließlich auch noch den Aktivierungszustand in eine **Ausgabefunktion** stecken, die entscheidet, welcher Wert an die Synapsen geschickt wird. In der Regel nimmt man hier aber keine Veränderungen vor, d.h. man wählt hier die **Identität**, also die Funktion f(x) = x, also eine Funktion die keine Veränderung vornimmt.

Grundlegendes Funktionsprinzip im Allgemeinen

Lebewesen haben Sinneszellen, die ihr Signal an das Gehirn weiterleiten können. Ein künstliches neuronales Netz hat (oft) ebenfalls eine **Eingabeschicht**, die man sich als "Sinneszellen" vorstellen kann, welche z.B. die Pixelwerte eines Bildes aufnimmt. Im Beispielmodell vom Anfang wären dies die Neuronen A und B.

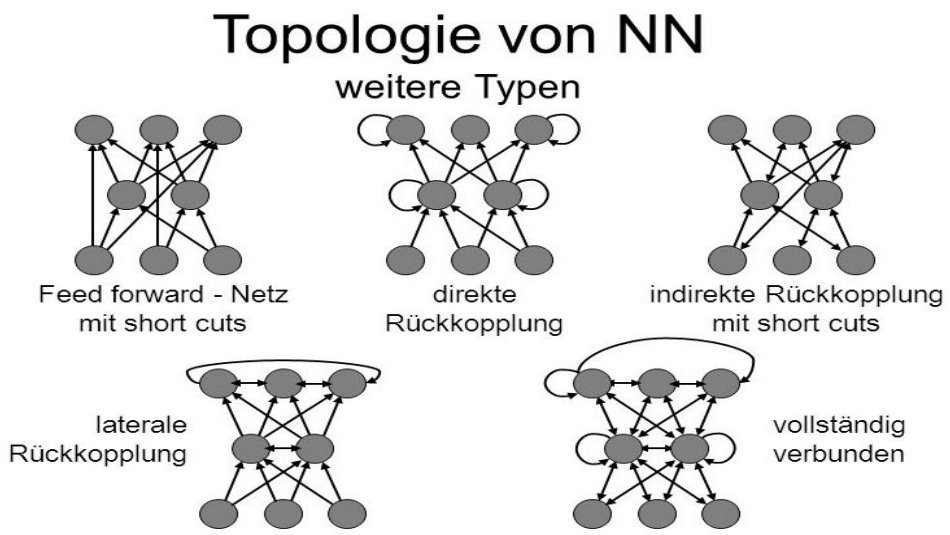
Indem die Aktivierungen der Zellen sich über das Netz ausbreiten, gelangen die Daten irgenwann an die **Ausgabeschicht** (im Beispiel das Ausgabeneuron X).

Oft baut man dazwischen sogenannte **Hidden Layer**, also weitere Verarbeitungsschichten.



Das Netz in der Abbildung hat vier Neuronenschichten, und damit drei **Gewichtsschichten**. Die Abbildung lässt schon erahnen, dass die Anzahl der Synapsen stark ansteigt, je mehr Neuronen dazukommen. Man kann zwar nicht so einfach pauschal sagen, dass ein neuronales Netz umso leistungsfähiger ist, je mehr Neuronen und Synapsen es hat, jedoch hat man in der Praxis meistens wesentlich mehr Elemente als nur ein paar Dutzend wie in den Abbildungen angedeutet.

Mathematisch ausgedrückt ist ein neuronales Netz nichts weiter als eine (komplizierte) Funktion, welche Eingabewerte auf Ausgabewerte abbildet, also eine Abbildung der Art $R^n \rightarrow R^m$, wobei n natürlich eine sehr große Zahl sein kann (wobei m möglicherweise sehr klein ist: Möglicherweise nur eine einzige Zahl, beispielsweise ein Gewissheitsindikator dafür, dass das Eingangssignal (z.B. ein Bild) ein Foto von Donald Trump ist).

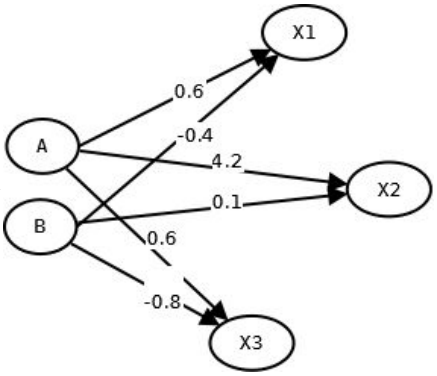


Unser Beispielnetz vom Anfang mit drei Neuronen hatte also keinen Hidden Layer. Selbstverständlich kann man sich noch weitere **Topologien** (also räumliche Strukturen) ausdenken (und je nach Anwendung wird das auch getan) – siehe Abb.

Falls ein Input immer weiter "nur nach rechts" durchgeschubst wird (wenn also keine Rückkopplungen eingebaut sind), so erhält man ein **Feed-Forward-Netz**. Diese sind besonders einfach zu programmieren und gut theoretisch untersuchbar.

Lernen
Die Fähigkeit eines neuronalen Netzes ist in den Gewichten gespeichert. Durch geschickte Änderung der Gewichte trainiert man das Netz so, dass es zu einem bestimmten Input (Aktivierung der Eingangsschicht) einen gewünschten Output (Aktivierung der Ausgabeschicht) liefert.
Eine Möglichkeit ist das **überwachte Lernen:** Man präsentiert dem Netz mehrere Eingaben zu denen man die gewünschte Ausgabe kennt. Liefert das Netz die falsche Ausgabe, "dreht" man etwas an den Gewichten. Wenn man genügend gutes Trainingsmaterial hatte (und die Netztopologie, die Aktivierungsfunktion etc. günstig gewählt hat) , kann das Netz dies nun generalisieren und auch bei neuen, unbekannten Eingaben richtige Ergebnisse erzeugen.

Konkrete Betrachtungen und Beispiele
Beispiel 1: Eine etwas ausführlichere Beispielrechnung. Wir nehmen an, dass das abgebildete Netz in der Eingangsschicht jeweils die Aktivierungszustände 1 besitzt, die Aktivierung der Eingangsschicht also durch den Vektor $\begin{pmatrix}1\\1\end{pmatrix}$ beschrieben werden kann. Darüberhinaus nehmen wir an, dass die Propagierungsfunktion bei jedem Neuron einfach (wie auf Seite 1 beschrieben) die gewichtete Summe berechnet. Die Aktivierungsfunktion soll (bei jedem Neuron) tanh sein, dieAusgabefunktion (bei jedem Neuron) die Identität.

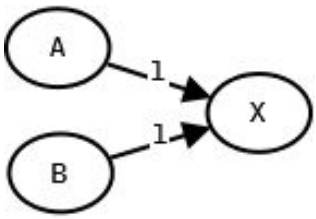


$$\begin{pmatrix}0{,}6 & -0{,}4\\ 4{,}2 & 0{,}1\\ 0{,}6 & -0{,}8\end{pmatrix}\cdot\begin{pmatrix}1\\1\end{pmatrix}=\begin{pmatrix}0{,}2\\ 4{,}3\\ -0{,}2\end{pmatrix}$$

Die Aktivierung von X1 liegt also nun bei tanh(0,2) ≈ 0,197
Die Aktivierung von X2 liegt also nun bei tanh(4,3) ≈ 0,9996
Die Aktivierung von X3 liegt also nun bei tanh(-0,2) ≈ -0,197

Wir stellen fest: Die Aktivierung von A hat also auf das Neuron X2 einen erhelblichen Einfluss.

Beispiel 2: Berechnung der logischen "ODER" Funktion.
Wir nehmen für das Neuron X folgendes an:
* Propagierungsfunktion ist die gewichtete Summe.
* Aktivierungsfunktion ist die Schwellenwertfunktion mit Schwellenwert 0.5.
* Ausgabefunktion ist die Identität.

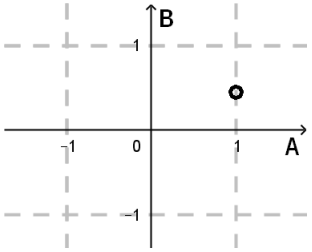


Eingabe (0 0) liefert bei X die Aktivierung 0
Eingabe (1 0) liefert bei X die Aktivierung 1
Eingabe (0 1) liefert bei X die Aktivierung 1
Eingabe (1 1) liefert bei X die Aktivierung 1

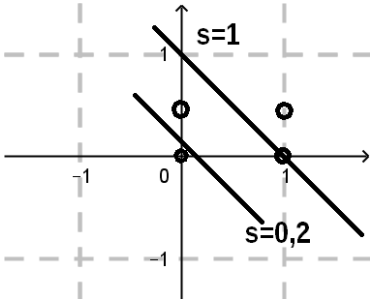
Damit hat man eine "ODER"-Operation modelliert. "X ist 1, wenn A oder B 1 ist".

Übung 1: Modelliere ein Netz, welches die "UND"-Operation modelliert.
Übung 2: Modelliere ein Netz, welches die "NOT"-Operation modelliert (Das "!" in Java)

Beispiel 3: Das XOR-Problem
Die XOR-Operation ("entweder oder – aber nicht beides!") verknüpft zwei Bits und liefert als Ergebnis eine 1, wenn genau eines der beiden Eingabebits 1 war – sonst ist das Ergebnis 0. Ein Versuch, diese Operation mit drei Neuronen, (mit der gewichteten Summe als Propagierungsfunktion und einer Schwellenwertfunktion als Aktivierungsfunktion) abzubilden, scheitert. Grund:
Man kann die Ausgaben der Neuronen A und B in einem Diagramm darstellen. Angenommen, die Gewichte sind jeweils 1 (für A) und 0,5 (für B) dann stellt markierte Punkt (1| 0,5), die Eingabe von X vor der Summenbildung dar – falls sowohl A als auch B eine Aktivierung von 1 besaßen. Falls eine der beiden Neuronen A oder B die Aktivierung von 0 hatte, so wäre der Punkt auf der y- bzw. der x-Achse. Wären beide Aktivierungen gleich Null gewesen, so ergäbe sich der Ursprung.



Die empfangende Zelle X legt nun einen Schwellwert nach der Summenbildung an. Dadurch wird eine Gerade im Koordinatensystem festgelegt. Alle Punkte "rechts über" der Geraden bilden eine Summe, die über dem Schwellwert liegt. In der Abbildung sind die Geraden für den Schwellwert 1 bzw. den Schwellwert 0,2 eingetragen.



Übung 3:
Zeichne das entsprechende Diagramm für die Gewichte -0,8 (A) und 1,2 (B) und dem Schwellwert s=0,5 und gib die Tabelle für die entstehende binäre Funktion an.

Man erkennt, dass die Gerade die vier Punkte in jeweils zwei Teile teilt: Diejenigen, die den Schwellwert überschreiten, und jene, die das nicht tun. Und jeder Punkt entspricht ja einem der vier Eingabemuster.
Das Problem beim XOR ist nun, dass man keine Gerade (d.h. keinen Schwellwert) finden kann, die die vier Punkte so teilt, dass nur die beiden Punkte auf den Koordinatenachsen (was ja dem Muster "nur genau ein Neuron ist aktiviert" entspricht) den Schwellwert überschreiten. Der Experte sagt: Das Problem ist nicht **linear separierbar**.
Es zeigt sich, dass man dieses Konzept erweitern kann und man kommt zum Schluss: Ein neuronales Feed-Forward-Netz ohne Hidden Layer (also nur eine Eingabe- und eine Ausgabeschicht) kann nur linear separierbare Probleme lösen.

Diese Erkenntnis hat im Jahr 1969 dazu geführt, dass man Forschungsgelder auf dem Gebiet der KI gestrichen hat, weil das Konzept als Ganzes nicht mehr vielversprechend erschien.

Allerdings kann man das Problem ganz einfach durch Einführung von Hidden Layers lösen (siehe Abbildung 5.9)

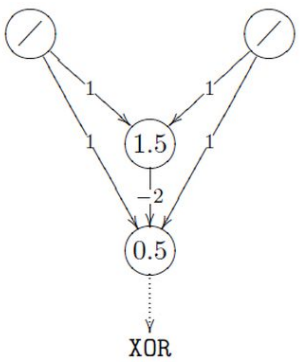
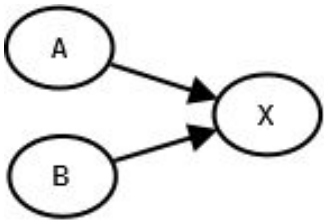


Abbildung 5.9: Neuronales Netz, welches die XOR-Funktion realisiert. Schwellenwerte stehen (soweit vorhanden) innerhalb der Neurone.

Training eines neuronalen Netzes: Die Delta-Regel



Wenn wir uns für eine Struktur eines Netzes entschieden haben, so möchten wir es auch trainieren. Die Frage ist, wie wir nun die Gewichte des Netzes geschickt modifizieren, um ihm etwas "beizubringen". Der Übersichtlichkeit halber wählen wir wieder ein ganz einfaches Netz mit drei Neuronen. Propagierungsfunktion ist die gewichtete Summe, Ausgangsfunktion ist die Identität, Aktivierungsfunktion ist ebenfalls die Identität.

Angenommen, wir möchten dem Netz beibringen, in der Ausgabe X dann eine hohe Aktivität zu zeigen, wenn die Aktivität von B größer ist als die von A.
Als Trainingsset wählen wir:

(0,1) soll (1) liefern,
(1,0) soll (-1) liefern.

Da wir zu Beginn noch gar nicht wissen, wie unsere Gewichte aussehen sollen, wählen wir eine zufällige Belegung: Gewicht $g_{AX} = 0{,}2$ und Gewicht $g_{BX} = 0{,}4$

Wir testen nun die beiden Trainingsdatensätze:

		Fehler:
Datensatz 1: $0{,}2 * 0 + 0{,}4 * 1 = 0{,}4$	aber eigentlich sollte 1 rauskommen!	$(0{,}4-1) = -0{,}6$
Datensatz 2: $0{,}2 * 1 + 0{,}4 * 0 = 0{,}2$	aber eigentlich sollte -1 rauskommen!	$(0{,}2 - (-1)) = 1{,}2$

Wir berechnen nun den Gesamtfehler (als quadratischer Fehler, vgl. z.B. Standardabweichung!):
Gesamtfehler $= (-0{,}6)^2 + (1{,}2)^2 = 1{,}8$

Übung 5: Berechne für obigen Trainingsdatensatz den Gesamtfehler für die Gewichte $g_{AX} = -0{,}1$ und Gewicht $g_{BX} = 0{,}6$

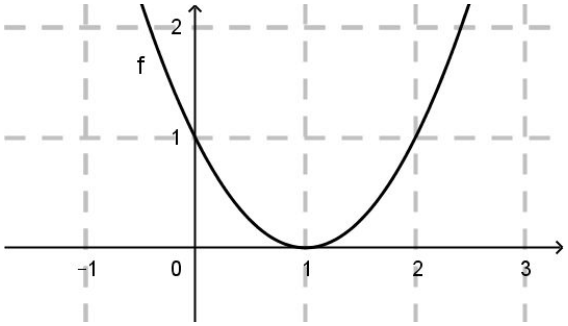
Hätten wir einen Gesamtfehler von Null, so wären wir fertig. Weil dem nicht so ist, müssen wir unsere Gewichte anpassen. Den Gesamtfehler kann man auch als Funktion auffassen, welche in Abhängigkeit von g_{AX} und g_{BX} eine reelle Zahl liefert:

Gesamtfehler(g_{AX} , g_{BX}) ist eine Abbildung $R^2 \rightarrow R$

Von dieser Funktion sucht man nun ein Minimum!
Aus der Analysis wissen wir, dass die erste Ableitung für dieses Problem sehr hilfreich ist. Dies ist der Grund dafür, dass man in der Regel differenzierbare (ableitbare) Aktivierungsfunktionen wählt! Die erste Ableitung einer Abbildung $R^2 \rightarrow R$ ist ein Vektor der keine zwei Zahlen, sondern zwei Funktionen enthält; er heißt **Gradient**.

Das Standardverfahren "Extremwertsuche" verlangt eigentlich zunächst die Nullstellen der ersten Ableitung. Leider ist es in der Praxis viel zu kompliziert, die Nullstellen des Gradienten zu bestimmen. Daher bedient man sich des sogenannten **Gradientenabstiegsverfahrens**. Man bestimmt die Ableitung an einem Punkt (in unserem Fall an dem Punkt ($g_{AX} = 0{,}2$, $g_{BX} = 0{,}4$)) erhält damit einen Vektor, der die (größte) "Steigung" der Funktion wiederspiegelt. Ändern wir unsere Gewichte ein wenig in die "Gegenrichtung", so wird der Gesamtfehler kleiner. Dies machen wir nun so oft, bis der Gesamtfehler unter eine bestimmte Schwelle sinkt.

Die Deltaregel im eindimensionalen Fall



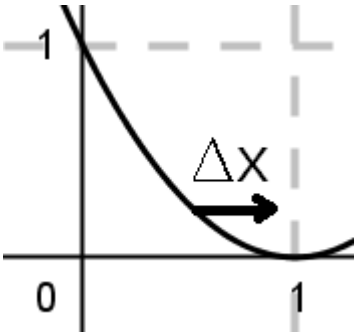
Da wir wenig Erfahrung mit Funktionen der Art $R^2 \rightarrow R$ haben, machen wir uns das Verfahren an der Funktion $f(x) = (x-1)^2 = x^2 - 2x + 1$ klar:

Der x-Wert ist hier nun die Entsprechung für ein einziges Gewicht und der y-Wert ist die Entsprechung für den Gesamtfehler.
Angenommen, unsere Synapse hat momentan das Gewicht $x = 0,5$, und man erhält den Gesamtfehler $f(0,5) = 0,25$.
Wir können $f'(x) = 2x - 2$ bestimmen und sehen: $f'(0,5) = -1$
Wenn man ausgehend von x=0,5 nun also (leicht) nach rechts wandert, so muss f(x) kleiner werden.
Wir berechnen zunächst die Intensität, um die wir den x-Wert verändern:

$$\Delta x = - c \cdot f'(x)$$

Die Konstante c legt fest, wie stark im allgemeinen die Änderung ausfallen soll. Wir wählen c=0,4.
Übrigens: "Δx" (sprich: Delta-x) ist der Namensgeber für diese Lernregel.
Der nächste x-Wert ergibt sich so: $x_{neu} = x_{alt} + \Delta x$

Wir erhalten mit unserem Zahlenbeispiel: $\Delta x = - 0,4 \cdot f'(0,5) = 0,4$
$x_{neu} = 0,9$



Falls der Gesamtfehler (jetzt: $f(0,9) = 0,01$) immer noch zu groß ist, wiederholen wir den Vorgang.
Eine erneute Anwendung der Regel liefert

$$\Delta x = - 0,4 \cdot f'(0,9) = -0,4 \cdot - 0,2 = 0,08$$
$$x_{neu} = 0,98$$

Die Regel sorgt also "automatisch" dafür, dass – wenn wir uns dem Minimum annähern, das Δx kleiner wird und wir nicht darüber "hinwegsausen".
Nun beträgt der Fehler $f(0,98) = 0,0004$

Wenn wir c ungeschickt wählen, kann es aber natürlich dennoch passieren, dass wir das Minimum "überschreiten"

Übung: Wende die Deltaregel zwei mal auf obige Funktion f(x) an, wobei der Startwert $x = 2,8$ und $c = 2$ sein soll.

Fazit:
Mit der Delta-Regel kann man anhand eines Trainingsdatensatzes ein neuronales Netz trainieren.
Dies klappt nur bei Feed-Forward-Netzen, welche nur eine Gewichtsschicht haben.
Hat man mehrere Gewichtsschichten, so benutzt man die Delta-Regel "mehrmals" hintereinander.
Dies Verfahren nennt man **Backpropagation**.