

# 1 Vorbild der KNN und das Prinzip eines künstlichen Neurons

## 1.1 Natürliche neuronale Netze

Das Besondere an KNN und was sie von anderen mathematischen Verfahren unterscheidet ist, dass sie ursprünglich dem menschlichen Gehirn und den natürlichen neuronalen Netzen nachempfunden sind. Mit dem natürlichen Vorbild haben sie genau genommen jedoch kaum zu tun, da es nach aktuellem Stand noch unmöglich ist, komplett und korrekt die Eigenschaften der natürlichen neuronalen Netze in künstlicher Form abzubilden. Daher wird die folgende Darstellung der natürlichen neuronalen Netze stark vereinfacht sein, sodass noch deutlicher wird, inwiefern sich die KNN an ihnen orientieren.

Das menschliche Gehirn gilt als das komplexeste und vielschichtigste Organ in der ganzen Neurobiologie. Bekannt zu seinem Aufbau ist, dass es geschätzt 100 Milliarden ( $10^{11}$ ) Neuronen bzw. Nervenzellen enthält, die alle zusammen ein Netzwerk bilden. Verknüpft sind die Nervenzellen mit ca.  $10^{14}$  -  $10^{15}$  synaptischen Verbindungen (Kramer 2009: S. 119-122). Eine Nervenzelle besteht aus einem Zellkörper, vielen Dendriten und einem Axon (siehe Abb. 1.1). Die Dendriten sind dafür zuständig Signale, die es von anderen Neuronen bekommt, aufzufangen und an den Zellkörper hinzuleiten. Im Zellkörper findet deren Verarbeitung statt. Wenn die Summe der empfangenen Signale, die die Nervenzelle von den anderen Nervenzellen bekommen hat, einen gewissen Schwellenwert der Erregung überschreitet, wird ein elektrischer Impuls, das Aktionspotenzial, gesendet. Es wird dann gesagt, dass die Nervenzelle *feuert* (Corves 2005). Über das Axon wird dieser zu einer Synapse weitergeleitet. Bei einer Synapse handelt es sich um eine Verbindungsstelle zwischen zwei Nervenzellen, wodurch die Übertragung eines elektrischen Impulses von einer Nervenzelle zu einer anderen ermöglicht wird. Der elektrische Impuls sorgt dafür, dass die andere Nervenzelle entweder gehemmt oder erregt wird.

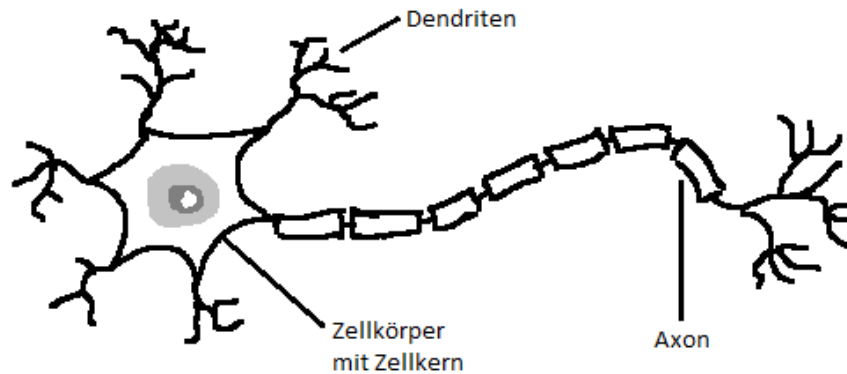


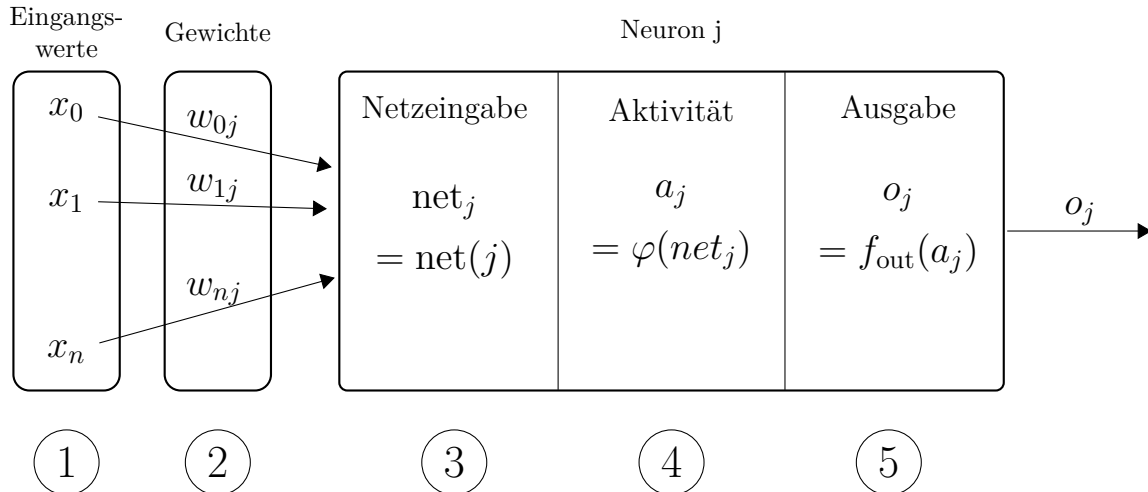
Abbildung 1.1: Aufbau einer Nervenzelle

## 1.2 Funktionsweise und Aufbau eines künstlichen Neurons

KNN bestehen aus einer technischen Version der Nervenzellen, den *künstlichen Neuronen*. Andere Bezeichnungen für Neuronen sind Units, Einheiten oder Knoten (Rey & Wender 1982: S.14). Ein einzelnes Neuron hat die einfache Aufgabe verschiedene Eingangssignale aufzunehmen, diese zu verarbeiten und daraus ein Ausgangssignal auszugeben. Zur Veranschaulichung der damit verknüpften mathematischen Zusammenhänge, soll die Abbildung 1.2 betrachtet werden. Die Nummerierung soll die Reihenfolge darstellen.

Es soll  $i, j, n \in \mathbb{N}$  gelten.

1. Ist  $n$  die Anzahl der Eingaben, so bekommt das Neuron  $j$  die Eingangssignale beziehungsweise *Eingangswerte*  $x_1$  bis  $x_n$  (biologische Analogie: Dendriten). Die Eingangswerte stammen entweder von anderen Neuronen oder Reizen aus der Umwelt.
2. Die *Gewichte* (engl. *weights*)  $w_{ij}$  stehen für die Stärke der Verbindungen (biologische Analogie: Synapse) vom Sender Neuron  $i$  zum Empfänger Neuron  $j$ . Durch sie wird bestimmt, zu welchem Grad die empfangenen Eingangswerte Einfluss auf die spätere Aktivierung des Neurons  $j$  nehmen werden. Je nach Vorzeichen werden die Eingangssignale jeweils verstärkt oder geschwächt.
3. Die *Netzeingabe*  $net_j$  verarbeitet die erhaltenen Informationen (also die Eingangssignale und die dazu gehörigen Gewichtungen) und berechnet sich mithilfe der Übertragungsfunktion  $net(j)$  (auch Propagierungs- oder Netzeingabefunktion genannt).



**Abbildung 1.2:** Funktionsweise eines Neurons, modifizierte Version aus <http://www.codeplanet.eu/tutorials/csharp/70-kuenstliche-neuronale-netze-in-csharp.html>

- Der *Aktivitätslevel*  $a_j$  beschreibt den aktuellen Zustand des Neurons  $j$  und berechnet sich aus der Netzeingabe mithilfe der Aktivierungsfunktion  $\varphi(net_j)$  (auch Aktivitätsfunktion genannt).
- Die *Ausgabe*  $o_j$  berechnet sich mithilfe der Ausgabefunktion  $f_{out}(a_j)$ . Sie ist der Wert, der entweder später anderen Neuronen als Eingabe gesendet wird oder einen Ausgabewert des KNN darstellt.

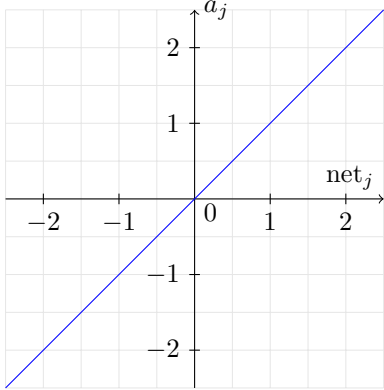
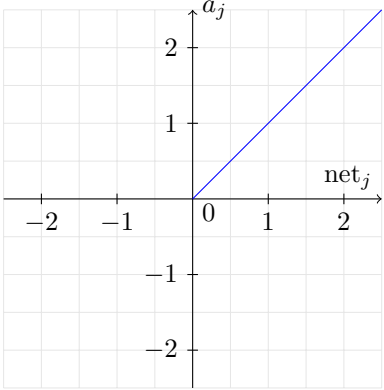
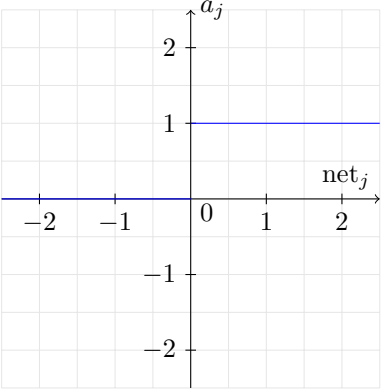
### 1.2.1 Übertragungsfunktion

Die am häufigsten verwendete Übertragungsfunktion ist die Linearkombination, bei der die Eingangssignale  $x_1$  bis  $x_n$  mit den Gewichten  $w_{1j}$  bis  $w_{nj}$  multipliziert werden und alle Produkte aufsummiert werden (siehe Gl. 1.1). Daher wird für die Übertragungsfunktion häufig als Symbol  $\sum$  verwendet.

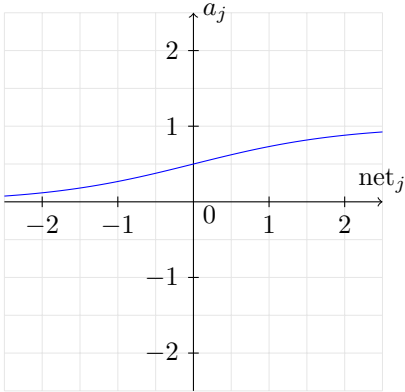
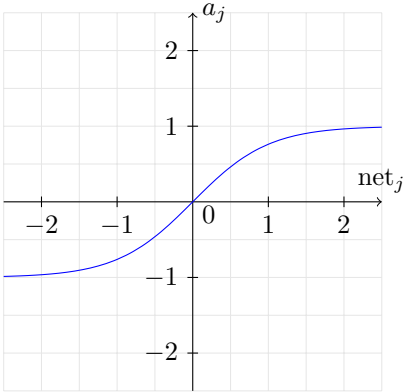
$$net_j = net(j) = \sum_{i=0}^n x_i w_{ij} \quad (1.1)$$

### 1.2.2 Aktivierungsfunktion

In der Literatur sind zahlreiche Beispielfunktionen zu finden, die sich als Aktivierungsfunktion nutzen lassen. In den Tabellen 1.1 und 1.2 sind die Aktivierungsfunktionen aufgeführt, die häufig zu finden sind.

Lineare Aktivierungsfunktion	ReLu (Rectified Linear Unit)	Binäre Aktivierungsfunktion
Linearer Zusammenhang zwischen dem Netinput $net_j$ und dem Aktivitätslevel $a_j$ . Weder nach unten noch nach oben ist der Wertebereich für den Aktivitätslevel beschränkt.	Lineare Aktivierungsfunktion mit Schwelle 0. Es ist erforderlich, dass die festgelegte Schwelle 0 überschritten wird, erst dann ist der Zusammenhang zwischen den beiden Größen linear.	Häufig wird diese Funktion auch Heavyside-Funktion, Schwellenwertfunktion oder Schrittfunktion genannt. Der Aktivitätslevel kann lediglich zwei Zustände annehmen, nämlich 0 (in manchen Fällen auch -1) oder +1.
$a_j = m \cdot net_j + b$	$a_j = \max(0, net_j)$	$a_j = \begin{cases} 1 & \text{falls } net_j \geq 0 \\ 0 & \text{falls } net_j < 0 \end{cases}$
 <p>mit <math>m = 1</math> und <math>b = 0</math></p>		

**Tabelle 1.1:** lineare Aktivierungsfunktion, ReLu und die binäre Aktivierungsfunktion

Logistische Aktivierungsfunktion	Tangens Hyperbolicus Aktivierungsfunktion
Der Wertebereich ist auf 0 bis +1 begrenzt. Werden große negative Werte (z.B. -100) für $net_j$ eingesetzt, ist der Aktivitätslevel nahe 0. Mit zunehmenden Netzinput steigt der Graph zunächst langsam, wird dann immer steiler (sodass er dabei zwischenzeitlich einer linearen Funktion gleicht) und nähert sich daraufhin wieder asymptotisch dem Wert +1 an.	Diese Funktion wird häufig abgekürzt tanh Aktivierungsfunktion genannt und verläuft ähnlich wie die logistische Aktivierungsfunktion. Der Unterschied besteht jedoch vor allem darin, dass der Wertebereich zwischen -1 und +1 liegt.
$a_j = \frac{1}{1+e^{-k \cdot net_j}}$	$a_j = \frac{e^{net_j} - e^{-net_j}}{e^{net_j} + e^{-net_j}}$
 <p>mit <math>k = 1</math></p>	 <p>mit <math>\theta_j = 0</math></p>

Dabei gilt:

$m$  = Steigung der Geraden

$b$  = Achsenschnittpunkt der Geraden

$k$  = konstanter Faktor

**Tabelle 1.2:** Logistische und tanh Aktivierungsfunktion

Öfters wird für die Aktivierungsfunktion (vor allem für die binäre Aktivierungsfunktion) ein bestimmter Schwellenwert  $\theta_j$  für das Neuron  $j$  mitberücksichtigt. Bevor das Neuron aktiviert wird, muss noch zusätzlich der Schwellenwert überschritten werden. Bei der Berechnung der Aktivierung kommt als zusätzlicher Schritt, dass der Schwellenwert von der Netzeingabe abgezogen wird (siehe Gl. 1.2).

$$a_j = \varphi(\text{net}_j - \theta_j) \quad (1.2)$$

Welche Aktivierungsfunktion gewählt wird, hängt von der Art des künstliche neurale Netze (KNN) bzw. vom konkreten Anwendungsproblem ab. Im Kapitel 3.5 soll nochmal darauf eingegangen werden.

### 1.2.3 Ausgangsfunktion

Bei natürlichen Neuronen ist es erforderlich, dass der Aktivitätslevel eine bestimmte Schwelle überschreitet, bevor das Neuron feuert. Diese Bedingung haben künstliche Neuronen nicht, dafür wird aber verlangt, dass der Ausgang mit zunehmender Aktivität nicht kleiner wird, d.h. die Ausgangsfunktion hat die Anforderung *monoton wachsend* zu sein. Für bestimmte Arten von KNN wie den hybriden Netzen ist eine klare Trennung zwischen der Aktivierungsfunktion und der Ausgangsfunktion erforderlich. In zahlreicher Literatur jedoch wird die Ausgangsfunktion von den Autoren als Bestandteil der Aktivierungsfunktion angesehen und daher gar nicht erwähnt. Im Grunde genommen wird dann also für die Ausgangsfunktion die Identitätsfunktion (siehe Gl. 1.3) eingesetzt, auch für diese Arbeit soll sie verwendet werden.

$$o_j = f_{\text{out}(a_j)} = a_j \quad (1.3)$$

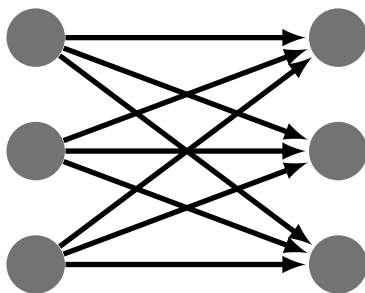
## 2 Struktur der KNN

### 2.1 Aufbaustruktur

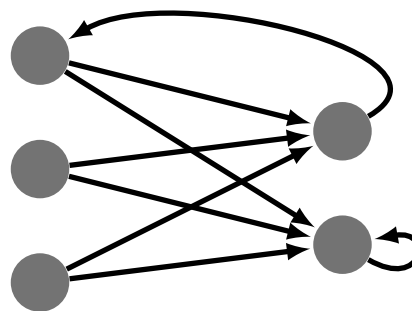
Der Aufbau eines KNN lässt sich als ein Netz künstlicher Neuronen beschreiben, die im Prinzip beliebig über gerichtete und gewichtete Verbindungen miteinander verknüpft sind. Als Netzwerkgraph kann das so aussehen, dass dessen Knoten die Neuronen und die Kanten die Verbindungen repräsentieren sollen. Aufgrund dieser Darstellungsmöglichkeit kann die mathematische Definition von KNN folgendermaßen lauten: Ein KNN „besteht aus einer Menge von Neuronen  $N = \{n_1, \dots, n_m\}$  und einer Menge von Kanten  $K = \{k_1, \dots, k_p\}$  mit  $K \subset N \times N$ “ (Scherer 1997: S.54).

Die Struktur und Anordnung der künstlichen Neuronen innerhalb des KNN wird als *Topologie* definiert. Welche Topologie verwendet werden soll, hängt vom konkreten Anwendungsproblem ab. Es lassen sich zwei Grundtypen von Netzen unterscheiden:

- Zum einen die *rückgekoppelten Netze* (bzw. *rekurrente Netzwerke* oder *Feedback Netzwerke*), bei denen die Ausgabewerte auf die Eingabe zurückgeführt werden können. Die Verbindungen können also in beide Richtungen verlaufen.
- Zum anderen die *vorwärts gerichteten Netze* (bzw. *Feedforward Netzwerke*), dessen berechneten Ausgabewerte keinen Einfluss auf die Eingabe haben. Hier können die Verbindungen also nur in eine Richtung gehen, nämlich von der Eingabe zur Ausgabe.



**Abbildung 2.1:** Schematisches Beispiel  
Feedforward Netzwerk



**Abbildung 2.2:** Schematisches Beispiel  
Feedback Netzwerk

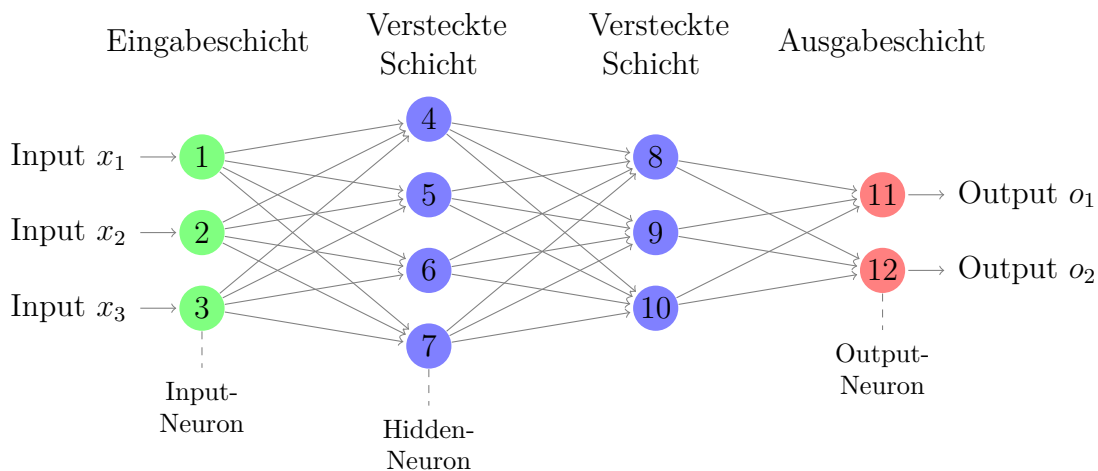
## 2 Struktur der KNN

Auf erstere soll nicht mehr näher eingegangen werden, zwar sind sie aufgrund ihrer größeren Leistungsfähigkeit im Vergleich zu den vorwärts gerichteten Netzen interessanter, doch dafür deutlich komplexer.

Bei den Feedforward Netzwerken können drei verschiedene Arten von Neuronen klassifiziert werden:

- Zum einen gibt es die *Input-Neuronen* (bzw. *Eingangsneuronen*), die Eingabesignale von der Außenwelt zum Beispiel in Form von Reizen und Mustern bekommen.
- Desweiteren gibt es die *Hidden-Neuronen*, die zwischen den Input- und Output-Neuronen stehen.
- Als letztes sind die *Output-Neuronen* (bzw. *Ausgangsneuronen*) aufzuführen, die die Aufgabe haben, Signale an die Außenwelt auszugeben und auszuwirken.

Häufig sind bei Feedforward Netzwerken die Neuronen in mehrere *Schichten* eingeteilt. Es gibt keine verbindlichen Regeln für die Einteilung eines Netzes in Schichten, für gewöhnlich werden die Neuronen zusammengefasst, die gemeinsam eine bestimmte Aufgabe durchführen. Beispielsweise bei der Abbildung 2.3 ist das Kriterium für die Einteilung die Verbindungen der Neuronen. Dabei lassen sich die Input-Neuronen 1,...,3 zur Eingabeschicht und die Neuronen 11 und 12 zur Ausgabeschicht zusammenfassen. Dazwischen bilden die Neuronen 4,...,7 und die Neuronen 8,...,10 jeweils eine versteckte Schicht.



**Abbildung 2.3:** Beispiel Feedforward Netzwerk, modifizierte Version aus <http://www.texample.net/tikz/examples/neural-network/>

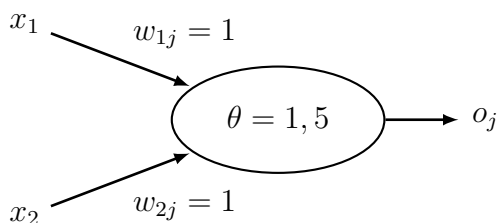


## 2.2 Einlagiges Perzeptron

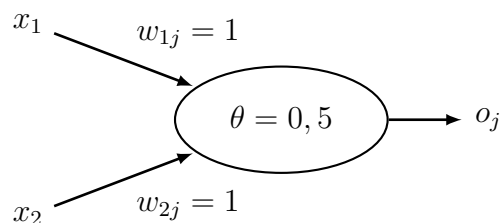
Die erste Fassung eines Perzeptrons wurde 1958 von Frank Rosenblatt vorgestellt. Eine vereinfachte Version davon, das *einfache Perzeptron*, das nur aus einem künstlichen Neuron besteht, ist als simpelstes Beispiel für die KNN zu nennen. Es zählt zu den *einlagigen Perzeptronen* (eng. *single-layer perceptron*). Das einlagige Perzeptron stellt ein Feedforward Netzwerk dar, das nur zwei Schichten, die Ein- und Ausgabeschicht enthält, die beide jeweils beliebig viele Neuronen enthalten können. Es gibt keine versteckte Schichten, daher sind alle Neuronen der Eingabeschicht mit denen der Ausgabeschicht direkt verbunden. Im folgenden Unterkapitel soll erläutert werden, was mit einfachen Perzeptronen unter anderem möglich ist.

### 2.2.1 Darstellung der booleschen Funktionen als einfaches Perzeptron

Mit dem bisher erworbenen Wissen lassen sich einfache boolesche Funktionen wie die Konjunktion, die Disjunktion und die Negation als einfache Perzeptronen darstellen. Es soll davon ausgegangen werden, dass als Übertragungsfunktion die Summenfunktion (siehe Gl. 1.1), als Aktivierungsfunktion die binäre Aktivierungsfunktion (siehe Tabelle 1.1) und als Ausgangsfunktion die Identitätsfunktion (siehe Gl. 1.3) genommen werden sollen. Die AND-Funktion und die OR-Funktion können dann zum Beispiel folgende Werte für die Gewichte und Schwellenwerte zugewiesen bekommen (siehe Abbildungen 2.4 und 2.5).



**Abbildung 2.4:** Repräsentation der Konjunktion  $x_1 \wedge x_2$  als einfaches Perzeptron



**Abbildung 2.5:** Repräsentation der Disjunktion  $x_1 \vee x_2$  als einfaches Perzeptron

Zum besseren Verständnis soll die Abbildung 2.4 genauer erläutert werden. Da es sich um eine boolesche Funktion handelt, werden für  $x_1$  und  $x_2$  nur Werte aus der Menge  $\{0, 1\}$  zugelassen. Um festzustellen, ob das Neuron  $j$  zum Beispiel für  $x_1 = 1$

und  $x_2 = 1$  feuert oder nicht, wird folgendes berechnet:

$$o_j = a_j = \varphi(\text{net}_j - \theta_j) \quad (2.1)$$

$$= \varphi((x_1 \cdot w_{1j} + x_2 \cdot w_{2j}) - \theta_j) \quad (2.2)$$

$$= \varphi((1 \cdot 1 + 1 \cdot 1) - 1,5) = \varphi(0,5) \quad (2.3)$$

Anhand der binären Aktivierungsfunktion sollte für  $o_j$  schließlich herauskommen:

$$o_j = \varphi(0,5) = 1 \quad (2.4)$$

Damit wurde festgestellt, dass das Neuron feuert. Bei den anderen möglichen Kombination der binären Eingabeparameter  $x_1$  und  $x_2$  sollte dagegen immer 0 herauskommen.

### 2.2.2 Klassifizierung und lineare Separierbarkeit

KNN werden häufig zur Klassifizierung genutzt. Dabei werden Klassen definiert, zu denen die verschiedenen Kombinationen der Eingabeparameter eingeordnet werden können. Im vorherigen Unterkapitel wurde also eingeteilt, welche Kombinationen der Eingabeparameter zur Klasse gehören, die das Neuron zum Feuern bringen und welche Kombinationen zu der Klasse, bei dem das Neuron dies nicht tut. Die *lineare Separierbarkeit* spielt für die Klassifizierung eine wesentliche Rolle. Unter linearer Separierbarkeit ist die Eigenschaft zu verstehen, zwei Mengen im  $n$ -dimensionalen Vektorraum durch eine *Hyperebene* voneinander trennen zu können. Die Dimension der Hyperebene beträgt  $(n - 1)$ , d.h. eine Dimension weniger als der Vektorraum, in dem es sich befindet. Beispielsweise ist im eindimensionalen Raum ein Punkt, im zweidimensionalen Raum die Gerade und im dreidimensionalen Raum die Ebene die Hyperebene.

Einlagige Perzeptronen sind in der Lage, separierbare Funktionen darzustellen, wobei die Dimension  $n$  sich aus der Anzahl der Eingabeparameter ergibt. Zur Veranschaulichung soll das Perzeptron bzw. Neuron aus Abbildung 2.4 geometrisch in einem zweidimensionalen Koordinatensystem interpretiert werden. Zweidimensional deshalb, weil es zwei verschiedene Eingabeparameter gibt. Dazu wird angenommen, dass für die Eingabeeinheiten nicht nur die Menge  $\{0, 1\}$  zugelassen wird, stattdessen soll  $x_1, x_2 \in \mathbb{R}$  gelten.

Da für die Funktion 2.2 die binäre Aktivierungsfunktion verwendet wird, muss folgende Ungleichung erfüllt werden, damit  $o_j = 1$  ergibt:

$$(x_1 \cdot w_{1j} + x_2 \cdot w_{2j}) - \theta_j \geq 0 \quad (2.5)$$

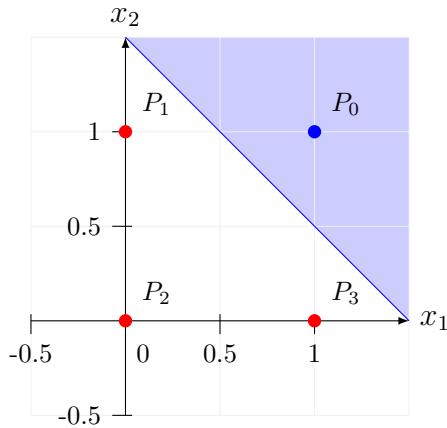
## 2 Struktur der KNN

Durch Äquivalenzumformungen ergibt sich daraus:

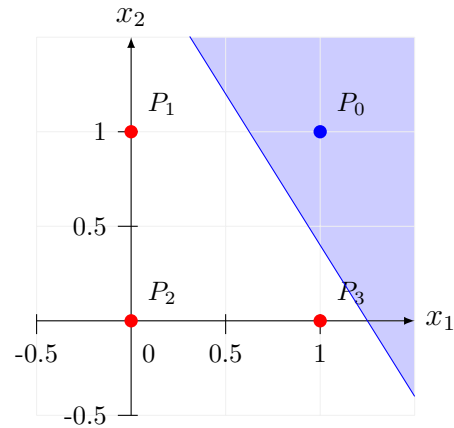
$$x_1 \geq -\frac{w_{1j}}{w_{2j}} \cdot x_1 + \frac{\theta}{w_{2j}} = \frac{1}{w_{2j}}(\theta - x_1 w_{1j}) \quad (2.6)$$

Unter der Annahme, dass  $x_1$  und  $x_2$  eine Ebene bilden, kann die Gleichung 2.6 als die sich in der Ebene ergebene Gerade bzw. Hyperebene sehen, die die Ebene in zwei Bereiche aufteilt. Oberhalb der Gerade befindet sich die Menge aller Punkte deren Kombination von  $x_1$  und  $x_2$  zu  $o_j = 1$  und damit zum Feuern des Neurons führen, sofern  $w_{2j}$  positiv ist. Logischerweise befindet sich dann unterhalb der Gerade die Menge aller Punkte, deren Kombinationen von  $x_1$  und  $x_2$  zu  $o_j = 0$  führen.

Die Abbildung 2.6 veranschaulicht grafisch die Trennung durch die Hyperebene. Für die Bildung der Geraden wurde die Geradengleichung 2.6 genommen und die aus Abb. 2.4 entnommenen Werte  $w_{1j} = 1, w_{2j} = 1$  und  $\theta = 1,5$  eingesetzt. Der blaue Bereich stellt den Bereich dar, in dem das Neuron feuert. Bei der Betrachtung der Punkte, die aus den binären Eingabemöglichkeiten resultieren, ist zu sehen, dass der Punkt  $P_0$  (1/1) in dem Bereich liegt, in dem das Neuron feuert und die Punkte  $P_1$  (0/1),  $P_2$  (0/0),  $P_3$  (1/0) in dem Bereich, in dem das Neuron nicht feuert. Bei der Abbildung 2.7 wurde in die Geradengleichung  $w_{1j} = 0.4, w_{2j} = 0.25$  und  $\theta = 0,5$  eingesetzt.



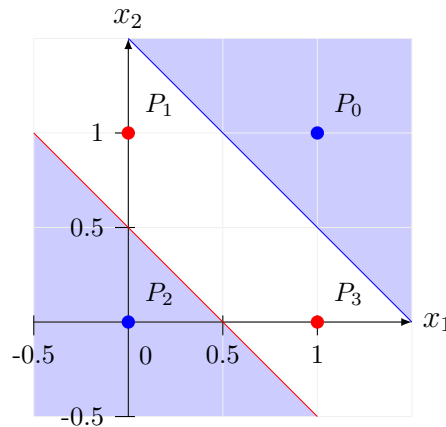
**Abbildung 2.6:** Beispiel 1 für die lineare Separierbarkeit der AND-Funktion



**Abbildung 2.7:** Beispiel 2 für die lineare Separierbarkeit der AND-Funktion

### 2.2.3 Das XOR-Problem

Geht es darum, Funktionen darzustellen, die nicht linear separierbar sind, stoßen die einlagigen Perzeptronen an ihre Grenzen.<sup>i</sup> So kann zum Beispiel die boolesche XOR-Funktion nicht als einlagiges Perzeptron repräsentiert werden. Wie die Abbildung 2.8 zeigt, reicht für korrekte Klassifizierung der Punkte  $P_1, \dots, P_4$  nur eine Hyperebene nicht aus und es werden mindestens zwei Hyperebenen benötigt. Es ist daher ein komplexeres KNN als das einlagige Perzeptron nötig, um das Problem zu lösen.



**Abbildung 2.8:** Die OR-Funktion ist nicht linear separierbar

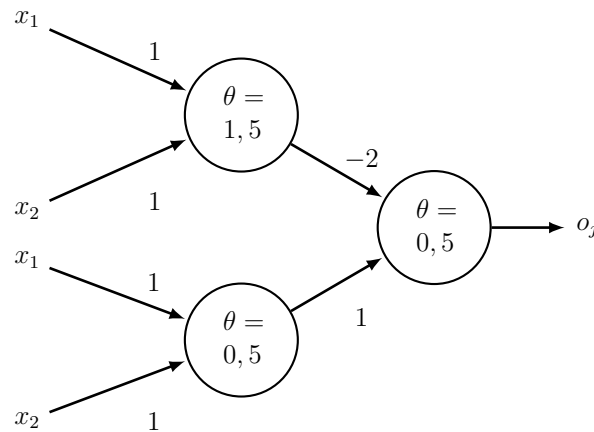
## 2.3 Mehrlagiges Perzeptron

Das mehrlagige Perzeptron (MLP) (auch als Multilayer Perzeptron bekannt) enthält neben der Ein- und Ausgabeschicht zusätzlich noch mindestens eine versteckte Schicht, wobei jede versteckte Schicht im Prinzip unendlich viele Neuronen enthalten kann. Das in Abbildung 2.1 dargestellte Feedforward Netzwerk zeigt wie ein MLP aussehen kann. Eine Eigenschaft des MLP ist, dass in der Eingabeschicht und in jeder versteckten Schicht jedes Neuron  $i$  jeweils mit einem bestimmten Gewicht  $w_{ij}$  zu jedem Neuron  $j$  der darauffolgenden Schicht verbunden sein muss. Das MLP ist in der Lage, Funktionen zu repräsentieren, die nicht linear separierbar sind und daher komplexere Problemstellungen zu lösen.

<sup>i</sup>Dies wurde von den Kritikern *Marvin Minsky* und *Seymour Papert* 1969 in deren Buch *Perceptrons* (Minsky & Papert 1969) beschrieben, das eine Reihe an mathematischen Beweisen von Perzeptronen enthält, die unter anderem die Grenzen dieser aufzeigen. Die daraus gewonnenen Erkenntnisse führten dazu, dass das Interesse an KNN deutlich abnahm und daher die Forschungsgelder dafür gestrichen wurden. Erst ab 1985 brachten neue Erkenntnisse wieder das Forscherinteresse an KNN deutlich zum Steigen.

### 2.3.1 Darstellung der XOR-Funktion als MLP

Um das XOR-Problem zu lösen, gibt es verschiedene Möglichkeiten wie das KNN aufgebaut sein kann, z.B. das XOR-Netzwerk stellt eine Lösungsmöglichkeit dar. Die hier vorgestellte Lösungsmöglichkeit soll durch ein MLP erfolgen, das folgendermaßen gebildet werden kann: Bei der Betrachtung der Abbildung 2.8 kann die blaue Hyperebene durch das Perzeptron aus Abbildung 2.4 und die rote Hyperebene durch das Perzeptron aus 2.5 dargestellt werden. Beide Perzeptronen lassen sich so kombinieren, dass deren Ausgaben von einem dritten Perzeptron als Eingangssignale empfangen werden. Dieser verarbeitet die Eingangssignale und bestimmt anschließend, ob das Ausgangssignal  $o_j$  den Wert 1 oder den Wert 0 zugewiesen bekommt. Für  $o_j$  soll erst 1 herauskommen, wenn vom Perzeptron der OR-Funktion das Ausgabesignal 1 und vom Perzeptron der AND-Funktion das Ausgabesignal 0 gesendet wird. Die Kombination dieser drei Perzeptronen ist in Abb. 2.9 dargestellt.



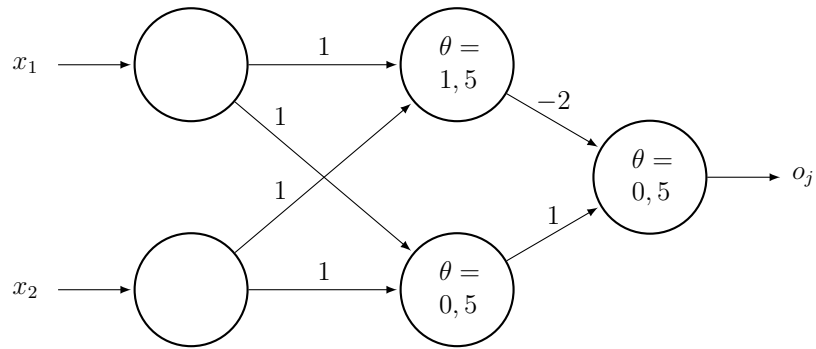
**Abbildung 2.9:** Durch die Kombination von drei Perzeptronen, kann die XOR-Funktion gelöst werden.

Zur besseren Übersicht werden häufig die Input-Neuronen verwendet, dabei ist für jedes Eingabesignal jeweils ein Input-Neuron zuständig. Da in diesem Beispiel zwei Eingabesignale  $x_1$  und  $x_2$  existieren, gibt es daher zwei Input-Neuronen. Ein Perzeptron, das in der Lage ist, das XOR-Problem zu lösen, kann mit Input-Neuronen aus fünf Neuronen bestehen und wie in Abbildung 2.10 aussehen.

### 2.3.2 Bias-Neuronen

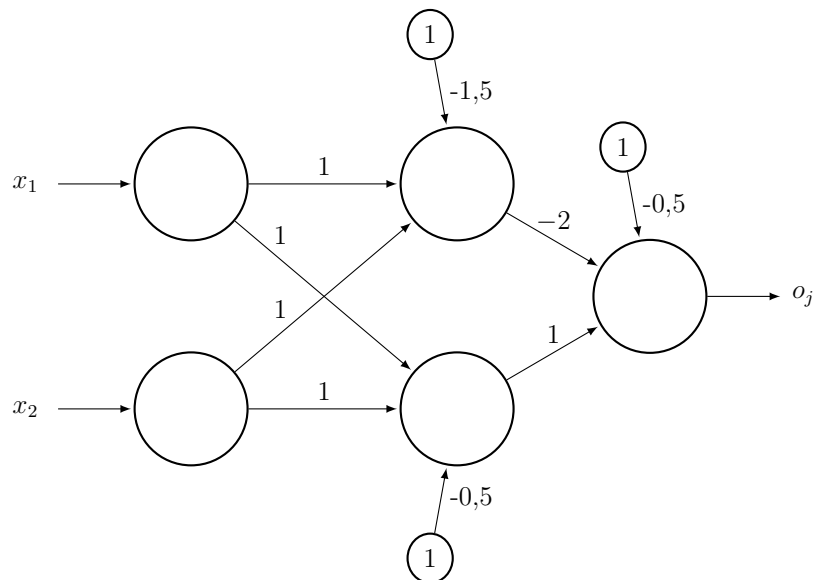
Das Bias-Neuron, auch on-Neuron genannt, ist eine Alternative, den Schwellenwert eines Neurons festzulegen. Er stellt einen weiteren Eingang  $x_0$  für das Neuron  $j$  dar und hat immer den konstanten Wert 1. Der negative Schwellenwert bildet dabei die Gewichtung, also  $w_{0j} = -\theta$ . Der praktikable Vorteil darin liegt, dass der Schwellenwert

## 2 Struktur der KNN



**Abbildung 2.10:** MLP, der die XOR-Funktion lösen kann ohne Bias-Neuronen

nicht mehr in der Aktivierungsfunktion enthalten ist, daher kann sie die Aktivierungsfunktion nicht mehr verändern. Zur Veranschaulichung der Bias-Neuronen kann die Abbildung 2.11 betrachtet werden.



**Abbildung 2.11:** MLP, der die XOR-Funktion lösen kann mit Bias-Neuronen

## 3 Lernen in KNN

In den vorherigen Beispielen wie Abbildung 2.4 oder Abbildung 2.10 sind bereits Netze mit den passenden Gewichten vorgegeben worden, die imstande sind, die zugehörige Problemstellung zu lösen. Das Interessante an KNN ist, dass sie meist zu Beginn noch nicht alle erforderlichen Informationen besitzen, um das Problem zu lösen. Stattdessen sind sie in der Lage selbständig aus Beispielen zu *lernen*, ein passendes Netz zur Lösung der Problemstellung zu bilden. In der Praxis wird der Vorgang des Lernens häufig auch als *Training* bezeichnet. Das Prinzip aus Beispielen zu lernen, entspricht dem des maschinellen Lernens. Es gibt viele mögliche Arten des Lernens bei KNN, zum Beispiel das Löschen existierender Verbindungen oder Neuronen oder auch das Entwickeln neuer Verbindungen oder Neuronen.

### 3.1 Lernen durch Modifikation der Gewichte

Vor allem hat es sich etabliert, das Lernen in KNN durch das Verändern der Werte für die Gewichte  $w_{ij}$  erfolgen zu lassen. Anhand einer vorher definierten *Lernregel* werden die Gewichte solange modifiziert, bis die Problemstellung zufriedenstellend gelöst werden kann. Um die neuen Gewichtswerte zu bestimmen, wird durch die Lernregel für jedes Gewicht  $w_{ij}$  ein Fehlerkorrekturwert  $\Delta w_{ij}$  bestimmt, der zum alten Gewichtswert hinzuaddiert wird. Als Formel ausgedrückt:

$$w_{ij}(\text{neu}) = w_{ij}(\text{alt}) + \Delta w_{ij} \quad (3.1)$$

### 3.2 Lernarten

Es lassen sich drei Lernarten von KNN unterscheiden, je nachdem wie die erwartete Ausgabe aussehen soll. Zur einfacheren Formulierung soll im Folgenden bei den verschiedenen Kombinationsmöglichkeiten der verschiedenen Eingabewerte von Eingabemustern und analog dazu bei den Ausgabewerten von Ausgabemustern gesprochen werden.<sup>i</sup>

---

<sup>i</sup>Veranschaulicht an einem Beispiel führt bei der AND-Funktion das Eingabemuster  $\{1, 1\}$  zum Ausgabemuster  $\{1\}$  und das Eingabemuster  $\{0, 1\}$  zum Ausgabemuster  $\{0\}$ .

- *Überwachtes Lernen* (engl. *supervised learning*): Die Trainingsdaten bestehen aus einer Menge an Paaren von Eingabe- und Ausgabemustern  $\{(E_p, O_p) | p \in \mathbb{N}^+\}$ , d.h. zu jedem Eingabemuster ist das erwünschte Ausgabemuster bekannt. Jedes Paar bzw. jeder Trainingsdatensatz kann als Muster (engl. pattern)  $p$  bezeichnet werden.
- *Bestärkendes Lernen* (engl. *reinforcement learning*) Hier besteht die Trainingsmenge aus einer Menge an Eingabemustern, zu denen jeweils nur bekannt gegeben wird, ob sie richtig oder falsch klassifiziert worden sind. Heißt also zu jedem aus dem Eingabemuster berechneten Ausgabemuster wird nur angegeben, ob das Ausgabemuster stimmt oder nicht. Wie das korrekte Ausgabemuster aussieht, muss das Netz selbst herausfinden.
- *Unüberwachtes Lernen* (engl. *unsupervised learning*) Bei dieser Art besteht die Trainingsmenge lediglich aus einer Menge von Eingabemustern. Dem Netz ist also unbekannt, welche Ausgabemuster herauskommen sollen und es entscheidet selber, welche Klassen sich aus den ausgerechneten Ausgabemustern ergeben. So können ähnliche Eingabemuster in ähnliche Klassen von Ausgabemustern eingeteilt werden.

Von den vorherigen vorgestellten Lernarten, ist das überwachte Lernen, die schnellste Art, das Netz zu trainieren. Die beiden anderen Lernarten sind dafür biologisch plausibler, da nicht die erwünschten Ausgabemuster (bzw. Aktivierungen) bekannt sein müssen.<sup>ii</sup> Deren Nachteile sind jedoch, dass sie deutlich langsamer sind als das überwachte Lernen.

## 3.3 Lernverfahren

Bei dem Lernprozess ist es weiterhin wichtig, das Verfahren zu bestimmen, welches festlegt, wann das KNN die Gewichte verändern soll:

- *batch- oder offline-Trainingsverfahren*: Bei diesem Verfahren muss dem System alle Muster  $p$  präsentiert werden, bevor alle Gewichte in einem Schritt modifiziert werden. Die Muster werden also stapelweise verarbeitet.
- *online-Trainingsverfahren*: Bei diesem Verfahren erfolgt die Änderung der Gewichte direkt nach Darbietung eines Musters  $p$  aus den Trainingsdaten.

---

<sup>ii</sup>Beispiel



## 3.4 Lernregel

Im Folgenden sollen Lernregeln vorgestellt werden, die vor allem für das überwachte Lernen eine Rolle spielen. Auf die genaue Herleitung der Regeln wird jedoch verzichtet, bei Interesse sei das Buch *Simulation neuronaler Netze* von *Andreas Zell* zu empfehlen (Zell 1994: S.105-110).

### 3.4.1 Hebbsche Lernregel

Die älteste und einfachste Regel, die für jede der drei Lernarten verwendet werden kann, ist die Hebbsche Lernregel. Anzumerken ist, dass sie lediglich auf KNN anwendbar ist, die keine versteckten Schichten besitzen. Dennoch ist sie aus dem Grunde interessant, weil es sich bei den darauf folgenden Lernregeln um Spezialisierungen der Hebbschen Lernregel handelt. Basieren tut sie in einer abgewandelten Form auf einer Vermutung vom Psychologen Donald Olding Hebb über die Veränderung von Synapsen in Nervensystemen (Hebb 1949). Die Regel besagt, dass wenn das Empfänger Neuron  $j$  vom Sender Neuron  $i$  eine Eingabe erhält und dabei beide gleichermaßen aktiv sind, dann das Gewicht  $w_{ij}$  erhöht wird (also die Verbindung von  $i$  nach  $j$  verstärkt). Die Formel für das offline-Trainingsverfahren soll der Einfachheit halber nicht gezeigt werden, stattdessen soll nur die für das online-Trainingsverfahren aufgeführt werden:

$$\Delta w_{ij} = \eta o_i a_j \quad (3.2)$$

Dabei gilt

- $\Delta w_{ij}$ : Änderung des Gewichts  $w_{ij}$
- $\eta$ : Lernrate, ein positiver konstanter Wert, der meist kleiner als 1 ist
- $o_i$ : Ausgabe der Sender-Neurons  $i$
- $a_j$ : Aktivierung des Empfänger-Neurons  $j$

Werden für  $o_i$  und  $a_j$  nur die binären Werte 0 und 1 zugelassen (wie ursprünglich bei der Hebbschen Regel angedacht), verändert sich  $w_{ij}$  nur, wenn beide den Wert 1 haben, bei den anderen Fällen findet keine Veränderung statt. Dann wird das Gewicht erhöht, eine Verringerung ist also nicht mehr möglich. Um dies zu umgehen, bietet es sich an, für  $o_i$  und  $a_j$  stattdessen die binären Werte -1 und +1 zuzulassen. In diesem Fall wird das Gewicht erhöht, wenn  $o_i$  und  $a_j$  identisch sind (jeweils +1 oder -1). Sind sie dagegen unterschiedlich, so werden die Gewichte verringert.

### 3.4.2 Delta-Regel

Die Delta-Regel (auch Widrow-Hoff-Regel genannt) kann für Netze verwendet werden, die das überwachte Lernverfahren nutzen. Die mathematische Formel für das online-Trainingsverfahren lautet:

$$\Delta w_{ij} = \eta o_i \delta_j \quad (3.3)$$

mit

$$\delta_j = t_j - a_j \quad (3.4)$$

Dabei gilt

- $t_j$ : teaching input, erwünschte Aktivierung
- $\delta_j$ : Differenz der aktuellen Aktivierung  $a_j$  und der erwünschten Aktivierung  $t_j$ .
- alle anderen Variablen sind analog zur vorher vorgestellten Hebb-Regel

Anhand der Differenz von erwünschter und berechneten Aktivierung wird ein Deltawert  $\delta_j$  bestimmt (siehe Gl. 3.4). Die Gewichtsänderung  $\Delta w_{ij}$  soll zur Größe des Fehlers  $\delta_j$  proportional sein (siehe Gl. 3.3).

Anwendbar ist die Delta-Regel ebenfalls nur bei KNN ohne versteckte Schichten, da nur die Ausgabewerte für die Output-Neuronen bekannt sind, aber nicht die für die Hidden-Neuronen. Dafür kann sie im Gegensatz zur Hebbschen Lernregel reellwertige Ausgaben erlernen.

### 3.4.3 Backpropagation-Regel

Mithilfe der Backpropagation-Regel ist es möglich, mehrlagige Perzeptronen zu trainieren. Bevor darauf eingegangen wird, wie die Backpropagation-Regel funktioniert, soll das *Gradientenabstiegsverfahren* vorgestellt werden, die für die Regel von Bedeutung ist.

#### Gradientenabstiegsverfahren

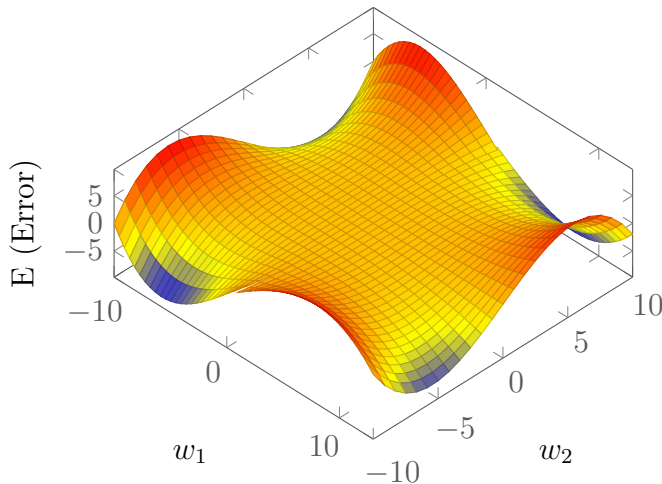
Um zu bestimmen, wie groß die Differenz zwischen erwarteter und tatsächlich berechneter Ausgabe ist, kann eine sogenannte Fehlerfunktion (siehe Gl. 3.5) bestimmt werden.

$$E(W) = E(w_1, \dots, w_n) \quad (3.5)$$

Es gilt

- $\mathbf{W}$  Gewichtsvektor, dessen Einträge aus  $w_1$  bis  $w_n$  bestehen

Anhand derer lässt sich bestimmen, wie groß der aufsummierte Fehler des KNN ist, wenn eine bestimmte Kombination der Gewichte  $w_1, \dots, w_n$  über alle Muster der Trainingsdaten erfolgt. Graphisch kann mit zwei Gewichten  $w_1$  und  $w_2$  die Funktion wie in Abbildung 3.1 dargestellt werden.



**Abbildung 3.1:** Bei zwei verschiedenen Gewichten kann die Fehlerfunktion  $E$  als Fläche dargestellt werden

Zur Veranschaulichung der Grafik wird oft folgendes Beispiel vorgestellt, um die Aufgabe des Gradientenabstiegsverfahrens klarzustellen: Eine Person befindet sich in einer Hügellandschaft, die so nebelig ist, dass er nur die unmittelbare Umgebung sehen kann. Ihr Ziel ist es, das tiefste Tal in der Landschaft zu finden. Um an ihr Ziel zu kommen, startet sie zunächst an einem beliebigen Startpunkt und prüft dort, in welcher Richtung es am steilsten bergab geht. Da sich die Person in einer 3-dimensionalen Landschaft befindet, muss sie sich daher komplett um die eigene Achse drehen, um den steilsten Abstieg zu finden. Sie folgt einige Meter dem steilsten Abstieg, prüft an der neuen Stelle, ob die Richtung des steilsten Abstiegs sich geändert hat und korrigiert ihre Laufrichtung gegebenenfalls danach. Dies wiederholt sie solange bis sie zu einer Stelle kommt, von dem es nicht mehr bergab geht oder bis sie zu erschöpft ist, um weiterzulaufen.

Das Beispiel zeigt eine ungefähre Vorstellung wie das Gradientenabstiegsverfahren für die KNN funktioniert. Das tiefste Tal stellt für den Fehlerterm das *globale Minimum* dar, um so die Gewichtungskombination zu finden, die zum geringsten Fehler zwischen erwarteter und tatsächlicher Ausgabe führt. Allgemein zeigt der sogenannte *Gradient* einer Funktion  $f$  im Punkt  $x$  in Richtung des steilsten Anstiegs einer Funktion  $f$ . Da aber in Richtung des steilsten Abstiegs gegangen werden soll, ist der negative Gradient der Fehlerfunktion (siehe Gl.3.5) zu bestimmen. Die Modifizierung der Gewichte,

um den Fehler zu minimieren, soll um einen Bruchteil des negativen Gradienten der Fehlerfunktion erfolgen (siehe Gl. 3.6). Übertragen auf das Beispiel soll damit festgelegt werden, wie weit die Person in Richtung des steilsten Abstiegs laufen soll.

$$\Delta W = -\eta \nabla E(W) \quad (3.6)$$

Es gilt

- $\eta$  Lernrate (bzw. Lernfaktor oder Schrittweite), die vorher festgelegt wurde; meist liegt sie zwischen 0,005 und 0,75
- $\nabla E(W)$ : Gradient der Funktion  $E(W)$  <sup>iii</sup>

Der Gradient der Fehlerfunktion ist im  $n$ -dimensionalen Raum ( $n$  = Anzahl der Gewichte + 1) ein  $(n-1)$ -dimensionaler Vektor. Die Einträge dieses Vektors bildet sich aus den ersten partiellen Ableitungen der Funktion. Der erste Eintrag ist die partielle Ableitung des ersten Gewichtes  $w_1$ , der zweite Eintrag der des zweiten Gewichtes  $w_2$  usw. Für die Änderung eines einzelnen Gewichtes  $w_{ij}$  gilt daher:<sup>iv</sup>

$$\Delta w_{ij} = -\eta \frac{\partial E(W)}{\partial w_{ij}} \quad (3.7)$$

Zur Bestimmung des Gesamtfehlers  $E$  werden die Fehler über alle Muster  $p$  aufsummiert (siehe Gleichung 3.8) und durch  $P$ , die Anzahl der Muster, geteilt.

$$E = \frac{1}{P} \sum_p^P E_p \quad (3.8)$$

Eine wichtige Information, um  $E$  zu bestimmen, ist die Wahl der *Kostenfunktion*, die für das gesamte KNN gilt. Für diese Arbeit soll als Kostenfunktion die *mittlere quadratische Abweichung*, abgekürzt *MSE* (aus dem Englischen *mean squared error*), benutzt werden, welche aus Gründen der Übersichtlichkeit, nicht in dieser Arbeit aufgeführt und erläutert werden soll. Die Wahl des MSE als Kostenfunktion führt dazu, dass zur Bestimmung von  $E_p$  sich folgende Gleichung ergibt:

$$E_p = \sum_j (t_{pj} - o_{pj})^2 \quad (3.9)$$

Es gilt

- $t_{pj}$ : erwartete Ausgabe vom Neuron  $j$  beim Muster  $p$
- $o_{pj}$ : tatsächliche Ausgabe vom Neuron  $j$  beim Muster  $p$

---

<sup>iii</sup>  $\nabla$  wird nabla ausgesprochen

<sup>iv</sup>  $\partial$  steht für die partielle Ableitung

Der Schritt der Modifizierung der Gewichte und dem Berechnen des Gesamtfehlers erfolgt solange, bis der Gradient  $\nabla = 0$  ist (Analogie: für die Person geht es ab der Stelle nicht mehr weiter bergab) oder bis eine bestimmte Anzahl an Iterationen/Wiederholungen dieses Schrittes erreicht wurde (Analogie: die Person ist zu erschöpft zum weitergehen). Beim Gradientenabstiegsverfahren ergeben sich viele Probleme (beispielsweise, dass oft nicht das globale, sondern das lokale Minimum gefunden wird oder dass gute Minima übersprungen werden können), teilweise gibt es schon Lösungsansätze für diese Probleme.<sup>v</sup>

## Der Backpropagation-Algorithmus

Das Problem der Delta-Regel ist, dass immer die erwünschte Ausgabe  $t$  bekannt sein muss, bei den Hidden-Neuronen sind diese aber in der Regel nicht gegeben. Um dieses Problem zu umgehen, wurde der Backpropagation-Algorithmus entwickelt, der die Trainingsphase/Lernphase jeder Gewichtsänderung in mehrere Schritte aufteilt:

1. Festlegen des Startpunktes: Der Startpunkt wird anhand einer zufällig ausgewählten Gewichtungskombination bestimmt.
2. Trainingsmuster auswählen
3. Forward-pass: Dem KNN werden Eingabemuster präsentiert und daraus die Ausgabemuster des Netzwerks berechnet. Von der ersten bis zu letzten versteckten Schicht sowie der Ausgabeschicht werden die Ausgaben zwischengespeichert.
4. Fehlerbestimmung: Es wird der Gesamtfehler  $E$  (siehe Gl. 3.8) zwischen erwünschter und tatsächlicher Ausgabe verglichen. Ist der Gesamtfehler gering (also überschreitet er keine gewisse Schwelle) oder ist die vorher festgelegte maximale Anzahl an Iterationen/Wiederholungen erreicht worden, kann die Trainingsphase abgebrochen werden, ansonsten wird zu Schritt 5 übergegangen.
5. Backward-pass: Durch die allgemeine (bzw. erweiterte) Delta-Regel (siehe Gl. 3.11) ist es möglich, sowohl für die Output- als auch für die Hidden-Neuronen ein  $\delta_{ij}$  zu bestimmen. Anders als beim Forward-pass ist die Reihenfolge umgekehrt, angefangen wird bei der Ausgabeschicht, dann geht es zur letzten versteckten Schicht, dann zur vorletzten usw. bis zur ersten versteckten Schicht. Durch dieses Vorgehen ist es möglich, nach und nach die Gewichte  $w_{ij}$  mithilfe des Gradientenabstiegsverfahren zu bestimmen, da die Gewichtsänderung unter anderem von  $\delta_{ij}$  abhängt. Dies wird durch die Formel für die Backpropagation-Regel deutlich:

$$\Delta_p w_{ij} = \eta o_{pi} \delta_{pj} \quad (3.10)$$

---

<sup>v</sup>Bei Interesse kann dies zum Beispiel bei (Rey & Wender 1982: S.43-48) nachgelesen werden.

mit

$$\delta_{pj} = \begin{cases} \varphi'(net_j)(t_{pj} - o_{pj}) & \text{falls } j \text{ ein Output-Neuron ist} \\ \varphi'(net_j) \sum_k \delta_{pk} w_{jk} & \text{falls } j \text{ ein Hidden-Neuron ist} \end{cases} \quad (3.11)$$

Es gilt

- $k$ : Das nachfolgende Neuron  $k$  befindet sich eine Schicht über der Schicht des Neurons  $j$

Alle Schritte ab Schritt 3 werden solange wiederholt, bis wie in Schritt 4 beschrieben, das Training unterbrochen werden kann.

Wie bereits erwähnt, verwendet die Backpropagation-Regel unter anderem das Gradientenabstiegsverfahren, deshalb ergeben sich dieselben Probleme und daher zahlreiche Varianten, die Backpropagation-Regel zu erweitern und zu modifizieren.

### 3.5 Auswahl der Aktivierungsfunktion

Binäre Aktivierungsfunktionen haben den Vorteil, dass sie schnell und einfach zu implementieren sind, aber dafür auch aufgrund ihrer mathematischen Eigenschaften nur für einfache Modelle von KNN in Frage kommen. Was ihnen fehlt, ist die Differenzierbarkeit an allen Stellen, was für die Backpropagation-Lernregel ein Problem ist. Dies ist bei genauerer Betrachtung der allgemeinen Delta-Regel 3.11 zu erkennen, die die Ableitung der Aktivierungsfunktion  $\varphi$  enthält. Differenzierbar und auch häufig eingesetzt sind sigmoide Aktivierungsfunktionen, zu denen die logistische Aktivierungsfunktion und die tanh Aktivierungsfunktion zählen. Der Trend geht jedoch immer mehr dazu über, die ReLu Funktion als Aktivierungsfunktion zu benutzen, die in der Praxis bei MLP mit sehr vielen versteckten Schichten effektiver ist.

# Abkürzungsverzeichnis

**MLP** mehrlagige Perzeptron

**KI** Künstliche Intelligenz

**KNN** künstliche neuronale Netze

**GUI** Grafical User Interface

# Abbildungsverzeichnis

1.1	Aufbau einer Nervenzelle . . . . .	2
1.2	Funktionsweise eines Neurons, modifizierte Version aus <a href="http://www.codeplanet.eu/tutorials/csharp/70-kuenstliche-neuronale-netze-in-csharp.html">http://www.codeplanet.eu/tutorials/csharp/70-kuenstliche-neuronale-netze-in-csharp.html</a> . . . . .	3
2.1	Schematisches Beispiel Feedforward Netzwerk . . . . .	7
2.2	Schematisches Beispiel Feedback Netzwerk . . . . .	7
2.3	Beispiel Feedforward Netzwerk, modifizierte Version aus <a href="http://www.texample.net/tikz/examples/neural-network/">http://www.texample.net/tikz/examples/neural-network/</a> . . . . .	8
2.4	Repräsentation der Konjunktion $x_1 \wedge x_2$ als einfaches Perzeptron . . .	9
2.5	Repräsentation der Disjunktion $x_1 \vee x_2$ als einfaches Perzeptron . . .	9
2.6	Beispiel 1 für die lineare Separierbarkeit der AND-Funktion . . . . .	11
2.7	Beispiel 2 für die lineare Separierbarkeit der AND-Funktion . . . . .	11
2.8	Die OR-Funktion ist nicht linear separierbar . . . . .	12
2.9	Durch die Kombination von drei Perzeptronen, kann die XOR-Funktion gelöst werden. . . . .	13
2.10	MLP, der die XOR-Funktion lösen kann ohne Bias-Neuronen . . . . .	14
2.11	MLP, der die XOR-Funktion lösen kann mit Bias-Neuronen . . . . .	14
3.1	Bei zwei verschiedenen Gewichten kann die Fehlerfunktion E als Fläche dargestellt werden . . . . .	19



# Tabellenverzeichnis

1.1	lineare Aktivierungsfunktion, ReLu und die binäre Aktivierungsfunktion	4
1.2	Logistische und tanh Aktivierungsfunktion . . . . .	5

# Literaturverzeichnis

- Bidelman, Eric: *Web Worker-Grundlagen*, <https://www.html5rocks.com/de/tutorials/workers/basics/>, 26.07.2010, letzter Zugriff: 29. 08. 2017
- Brij: *Concurrency vs Multi-threading vs Asynchronous Programming : Explained*, <https://codewala.net/2015/07/29/concurrency-vs-multi-threading-vs-asynchronous-programming-explained/>, 29.07.2015, letzter Zugriff: 29. 08. 2017
- Buckler, Craig: *Implementing JavaScript Threading with Web Workers*, <https://www.safaribooksonline.com/blog/2013/10/24/implementing-javascript-threading-with-web-workers/>, 24.10.2013, letzter Zugriff: 29. 08. 2017
- Corves, Anna: *Nervenzellen im Gespräch*, <https://www.dasgehirn.info/grundlagen/kommunikation-der-zellen/nervenzellen-im-gespraech?gclid=C0iWg7TdzNQCFU4W0wodK74IwA>, 2012, letzter Zugriff: 1. 07. 2017
- Ertel, Wolfgang: *Grundkurs Künstliche Intelligenz : Eine praxisorientierte Einführung*, 4. Aufl., Berlin Heidelberg New York: Springer-Verlag, 2016
- Hebb, Donald Olding: *The Organization Of Behavior: A Neuropsychological Theory*, Psychology Press edition 2002, 1949
- Hillmann, Leonard: *Intelligentes Leben*, <http://www.tagesspiegel.de/themen/gehirn-und-nerven/gesund-leben-intelligentes-leben/13410564.html>, Veröffentlichungsdatum unbekannt, letzter Zugriff: 01.07.2017
- Hoffmann, Norbert *Kleines Handbuch Neuronale Netze : Anwendungsorientiertes Wissen zum Lernen und Nachschlagen*, Berlin Heidelberg New York: Springer-Verlag, 1993
- Koenecke, Finn Ole: *Realisierung eines interaktiven künstlichen neuronalen Netzwerks*, 2016
- Kramer, Oliver: *Computational Intelligence : Eine Einführung*, 1. Aufl., Berlin Heidelberg New York: Springer-Verlag, 2009.
- Manhart, Klaus: *Was Sie über Maschinelles Lernen wissen müssen*, <https://www.computerwoche.de/a/was-sie-ueber-maschinelles-lernen-wissen-muessen,3329560>, 04.05.2017, letzter Zugriff: 02.07.2017

- Minsky, Marvin; Papert, Seymour: *Perceptrons: An Introduction to Computational Geometry*, 2nd edition with corrections, first edition 1969 , The MIT Press, Cambridge MA, 1972.
- Rey, Günter Daniel ; Wender, Karl F.: *Neuronale Netze : eine Einführung in die Grundlagen, Anwendungen und Datenauswertung*, 2. vollst. überarb. und erw. Aufl., Bern: Huber, 2011.
- Riley, Tonya: *Artificial intelligence goes deep to beat humans at poker*, <http://www.sciencemag.org/news/2017/03/artificial-intelligence-goes-deep-beat-humans-poker>, 03.03.2017, letzter Zugriff: 07.07.2017
- Rimscha, Markus: *Algorithmen kompakt und verständlich : Lösungsstrategien am Computer*, 3. Aufl., Berlin Heidelberg New York: Springer-Verlag, 2014.
- Russell, Stuart ; Norvig, Peter: *Artificial Intelligence : A Modern Approach*, 3. Aufl(2010), London: Prentice Hall, 2010.
- Scherer, Andreas: *Neuronale Netze : Grundlagen und Anwendungen*. Berlin Heidelberg New York: Springer-Verlag, 1997.
- SethBling: *MarI/O - Machine Learning for Video Games*, <https://www.youtube.com/watch?v=qv6UVOQOF44>, 13.06.2017, letzter Zugriff: 07.07.2017
- Turing, Alan M.: *Computing Machinery and Intelligence*, Mind, 59, 1950.
- Wolf, Jürgen: *C von A bis Z : das umfassende Handbuch*, [http://openbook.rheinwerk-verlag.de/c\\_von\\_a\\_bis\\_z/026\\_c\\_paralleles\\_rechnen\\_003.htm](http://openbook.rheinwerk-verlag.de/c_von_a_bis_z/026_c_paralleles_rechnen_003.htm) Bonn: Rheinwerk Verlag GmbH, 2009, letzter Zugriff: 29.08.2017
- Wunderlich-Pfeiffer, Frank : *Alpha Go geht in Rente*, <https://www.golem.de/news/kuenstliche-intelligenz-alpha-go-geht-in-rente-1705-128059.html>, 29.05.2017, letzter Zugriff: 04.07.2017
- Zell, Andreas: *Simulation neuronaler Netze*, 4.Auflage(2003) Deutschland: Oldenbourg Wissenschaftsverlag GmbH, 1994.