

Silas Hoffmann, inf103088

5. Fachsemester

6. Verwaltungssemester

16. April 2020

Thema 2

Seminar

im Sommersemester 2020

Dozent: Prof. Dr. Dennis Säring
Fachbereich Informatik

Fachhochschule Wedel

Inhaltsverzeichnis

I. Geschichtliche Entwicklung	4
1. McCulloch-Pitts-Neuron	4
1.1. Funktionsweise	4
1.2. Nachteile bzw. Verbesserungspotenzial	7
2. Perceptron	8
2.1. Aufbau und Notation	8
2.2. Lernregel	10
3. Adeline	12
3.1. Aufbau	12
3.2. Lernalgorithmus	12
4. Convolutional Neural Network	21
4.1. Geschichte	21
4.2. Funktionsweise	22
5. Fazit	27
5.1. Beantwortung der Leitfragen	27
II. Aktuelle Entwicklung	28
6. Backpropagation	28
6.1. Notation	28
6.2. Fundamentale Gleichungen	30
6.3. Anwendung	35
7. Multilayer Perceptron	37
8. Recurrent Neural Network	39
9. Fazit	40
9.1. Beantwortung der Leitfragen	40
A. Anhang	ii
A.1. McCulloch-Pitts-Zelle	ii
B. Architekturen	iv
B.1. Artenübersicht	iv

Abbildungsverzeichnis

1.	McCulloch-Pitts-Zelle - Genereller Aufbau	5
2.	McCulloch-Pitts-Zelle: Aufbau und Klassifizierung	6
3.	MPZ - Notation	6
4.	McCulloch-Pitts-Zelle - AND Gatter	6
5.	McCulloch-Pitts-Zelle - OR Gatter	7
6.	Perceptron - drei Eingabewerte	8
7.	Perceptron - Modelansicht	9
8.	Perceptor - Problematische Klassifizierung	11
9.	Adeline - Aufbau	12
10.	Gradientenverfahren mit eindimensionaler Kostenfunktion	13
11.	Gradientenverfahren - unterschiedliche Lernraten	14
12.	Gradientenverfahren - zweidimensionaler Eingabevektor	15
13.	Partielle Ableitung 1	16
14.	Partielle Ableitung 2	17
15.	Gradientenverfahren - Vergleich	20
16.	Vergleich - Simple und Complex Cell	22
17.	Vergleich - Darstellung Mensch und Maschine	23
18.	Überblick - CNN Verarbeitungsschritte	23
19.	CNN - Convolution Layer	24
20.	CNN - Filter	24
21.	Notation	29
22.	Notation	29
23.	MLP - Aufbau	38
24.	Sigmoid - Plot	38
25.	RNN - Aufbau	39
26.	McCulloch-Pitts-Zelle - OR Gatter 2	ii
27.	McCulloch-Pitts-Zelle - OR Gatter 3	ii
28.	McCulloch-Pitts-Zelle - Tautologie	iii
29.	Übersicht - verschiedene Architekturen	iv

Teil I.

Geschichtliche Entwicklung

Die Entwicklung künstlicher Intelligenz ist momentan eines der wohl interessantesten Themen der heutigen technologischen Ära. Durch *Machine Learning* ist es möglich viele Bereiche zu automatisieren die man dafür vorher nie in Betracht gezogen hatte. Dennoch hat es mich überrascht, wie „alt“ die Grundzüge dieser Technologie bereits sind. Die Anfänge finden sich schon Mitte der 50er Jahre als zum Beispiel diverse Experten auf diesem Gebiet das erste Mal zusammentrafen um im Zuge der *Dartmouth Konferenz* über das Thema zu diskutieren. Aber auch bekannte Persönlichkeiten des Feldes, wie zum Beispiel *Von Neumann*¹ erkannten früh die möglichen Zusammenhänge zwischen Mensch und Maschine.

Im folgenden Abschnitt werde ich etwas auf die geschichtlichen Aspekte von neuronalen Netzen eingehen. Hierbei werden insbesondere die generellen Aspekte der Funktionsweise von älteren Modellen bis hin zur aktuellen Entwicklung verfolgt. Ich werde versuchen die folgenden Leitfragen in diesem Abschnitt zu beantworten:

- Woher kommt Deep Learning und wie ist dieser Begriff im Kontext zur künstlichen Intelligenz einzuordnen?
- Welche Entwicklungen hat das Neuronale Netz von damals zu heute durchgemacht?

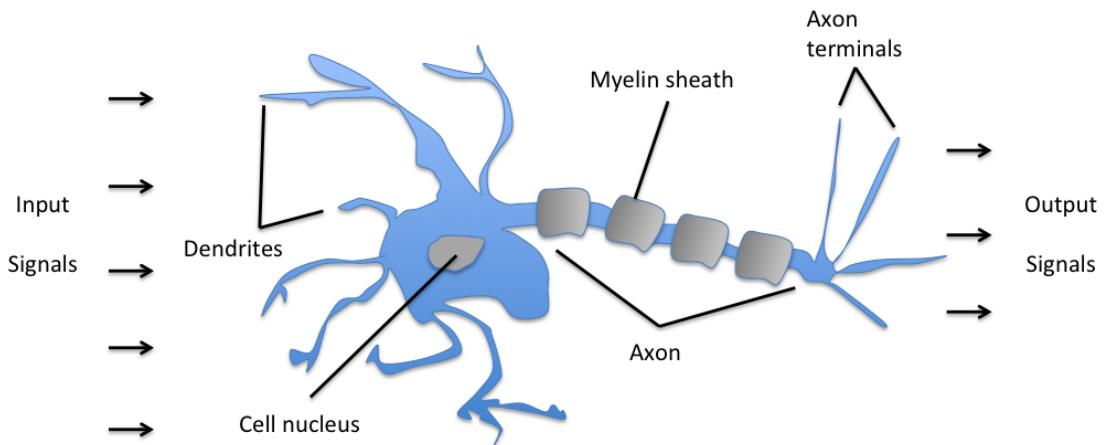
1. McCulloch-Pitts-Neuron

1.1. Funktionsweise

Im Jahr 1943 entwickelten Warren McCulloch und Walter Pitts ein Modell, das die Funktionalität eines biologischen Neurons imitieren sollte. In der folgenden Abbildung 1 ist der grobe Aufbau eines biologischen Neurons zu sehen.

Die sogenannten *Dendriten* (englisch *dendrites*) nehmen Informationen auf. Sie besitzen Rezeptoren, welche in der Lage sind Signale anderer Neuronen aufzunehmen. Diese Signale bewirken elektrische Veränderungen in dem Neuron, welche vom Zellkörper (*Soma*) interpretiert / verarbeitet werden. Dieser Zellkörper sammelt alle Informationen und speichert diese im sogenannten *Axonhügel* (engl. Axonhillock), welcher die Ursprungsstelle des *Axons* beziehungsweise der *Neuriten* beschreibt. Wenn das gebündelte Signal stark genug sein sollte, wird es an den nächsten Teil des Neurons (dem *Axon*) weitergeleitet. Ab diesem Zeitpunkt wird das Signal als *Aktionspotential* bezeichnet und wird über das *Axon* übertragen. Am Ende wird das Signal an diverse *Axonterminale* weitergeleitet, welche per Neurotransmitter mit den jeweils nächsten Dendriten verbunden sind.

¹Von Neumanns letztes Werk: zweiteiliges Manuskript über die damaligen Rechner und ihre Möglichkeiten das neuronale System zu emulieren.



Schematic of a biological neuron.

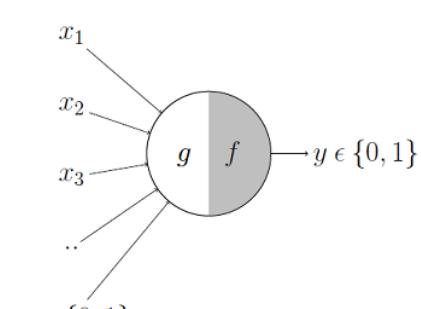
Abbildung 1: McCulloch-Pitts-Zelle - Genereller Aufbau [13]

Dieser biologische Aufbau dient als Grundlage für die Entwicklung des Modells von McCulloch und Pitts. Das Augenmerk ihres Modells liegt in erster Linie darauf Klassifizierungsprobleme zu lösen. Bei einem Klassifizierungsproblem wird *das zu klassifizierende Objekt X dabei durch einen Merkmalvektor \vec{x} aus dem betrachteten Merkmalraum M mit der Dimension n charakterisiert. Das Problem besteht nun darin zu entscheiden, ob das Objekt X in der betrachteten Klasse K liegt.* [3]. Der grobe Aufbau einer sogenannten *McCulloch-Pitts-Zelle* ist in Abbildung 3 zu sehen. Erwähnt sei ebenfalls, dass man mit diesem Modell lediglich binäre Klassifizierungen mittels einer linearen Entscheidungsfunktion / Aktivierungsfunktion durchführen kann (siehe Abbildung 2)

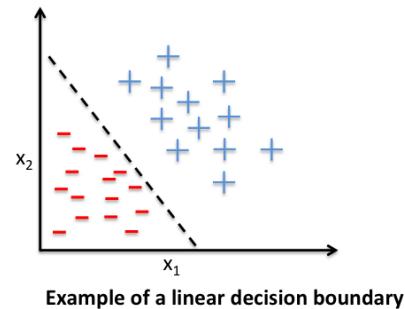
Das Modell kann beliebig viele Eingabe-Werte aufweisen. Wichtig hierbei: Sie dürfen nur boolescher Natur sein (nur falsch oder wahr). Bei gegebenen Werten führt das Neuron selbst zwei Arbeitsschritte durch:

1. Erst werden alle Werte aufaddiert (in der Abbildung dargestellt durch die Funktion g). Dies imitiert das Verhalten der *Dendriten* in einem biologischen Neuron.
2. Anschließend überprüft die Funktion f ob ein gegebener Schwellwert überschritten wurde oder nicht (gibt dies entsprechend in Form einer booleschen Ausgabe weiter). Das biologische Neuron tut dies mittels des *Axonhügels*.

Die übliche Notation dieses Modells gibt vor, dass der jeweilige Schwellwert jeweils in die linke Seite des Kreises geschrieben, während die rechte Seite ausgegraut wird. Im folgenden seien einmal beispielhaft das *AND* und das *OR* Gatter dargestellt. (Für weitere Beispiele siehe Unterabschnitt A.1)



(a)



**Example of a linear decision boundary
for binary classification.**

(b)

Abbildung 2: McCulloch-Pitts-Zelle: Aufbau und Klassifizierung [11]

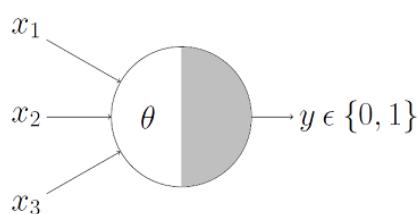


Abbildung 3: MPZ - Notation [11]

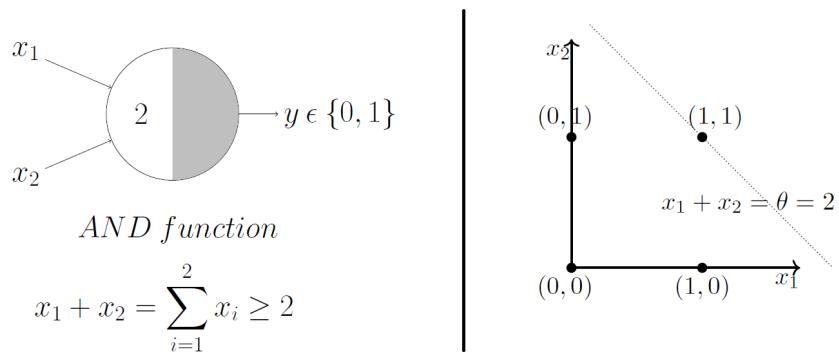


Abbildung 4: McCulloch-Pitts-Zelle - AND Gatter [11]

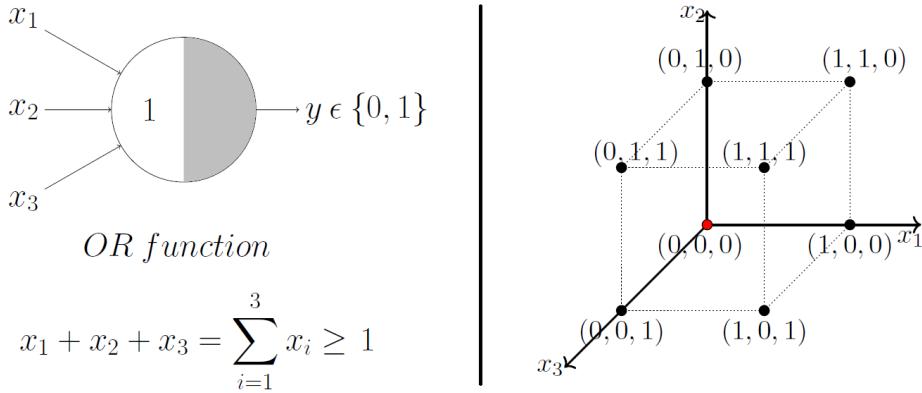


Abbildung 5: McCulloch-Pitts-Zelle - OR Gatter [11]

1.2. Nachteile bzw. Verbesserungspotenzial

- Dieses Modell erlaubt nur boolesche Eingabewerte, viele Modelle erfordern allerdings kontinuierliche Werte. Mit diesen wäre es zum Beispiel deutlich einfacher ein Bild oder Ähnliches zu analysieren.
- Die Schwelle (Theta) muss stets manuell bestimmt werden. Einen Trainingsalgorithmus, wie man ihn von heutigen Ansätzen her kennt, gibt es in diesem Modell nicht.
- Es gibt keinerlei Priorisierungsmöglichkeiten zwischen den einzelnen Eingabewerten. Jeder hat einen gleichgroßen Einfluss auf das Ergebnis, so etwas wie ein Ausschlusskriterium gibt es hierbei also nicht.
- Es ist nicht möglich *gedeckelt* Gatter wie zum Beispiel ein *XOR* abzubilden. Bei einem Neuron mit zwei Eingabe-Werten müsste zum Beispiel ein Schwellwert von 1 genau getroffen werden. Dieses Modell ist allerdings nur in der Lage zu entscheiden ob ein Schwellwert *überschritten* wurde oder nicht. Nichtlineare Entscheidungsfunktionen sind ebenfalls nicht möglich, die entstandenen Cluster müssen klar abtrennbar sein.

2. Perceptron

2.1. Aufbau und Notation

Im Jahr 1958 entwickelte der US-amerikanische Psychologe und Informatiker Frank Rosenblatt das sogenannte *Perceptron*. Dieses stellt das älteste neuronale Netz dar, welches teilweise auch heutzutage noch genutzt wird. Inspiriert wurde Rosenblatt vom Auge einer Fliege, wobei die Entscheidung der nächsten Flugrichtung in Teilen bereits im Auge stattfindet. Das Perceptron stellt in diesem Zusammenhang eine direkte Abbildung dieser Beobachtung dar.

Das Modell ist eine Weiterentwicklung von der McCulloch-Pitts-Zelle (siehe Abschnitt 1). Allerdings ist das Perceptron in der Lage die unterschiedlichen Eingabewerte zu priorisieren. Dies geschieht mittels reeller Gewichte, welche jeweils mit den Inputwerten verrechnet werden: $\sum_j w_j x_j$. Gleich bleibt jedoch die binäre Klassifizierung, wie schon bei der McCulloch-Pitts-Zelle.

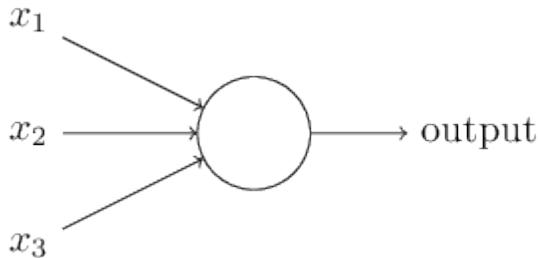


Abbildung 6: Perceptron - drei Eingabewerte [14]

Die Abbildung 6 beschreibt ein einfaches Perceptron mit drei Eingabewerten. Genau wie die MP-Zelle verwendet dieses Modell einen Schwellwert θ um den letztendlichen Ausgabewert zu bestimmen, jedoch möchte ich hier noch einmal etwas genauer auf die Notation des Ganzen eingehen, da diese auch in späteren Abschnitten benötigt wird.

Die Funktion, welche berechnet, ob ein Schwellwert überschritten wird oder nicht, wird *Aktivierungsfunktion* genannt (hier mit g bezeichnet).

$$g(\mathbf{z}) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{if } z < 0 \end{cases} \quad (1)$$

wobei gilt

$$\mathbf{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \quad (2)$$

$$\mathbf{z} = w_1 x_1 + \cdots + w_m x_m = \sum_{j=1}^m x_j w_j = \mathbf{w}^T \mathbf{x} \quad (3)$$

Sämtliche Gewichte und Eingabewerte können als Vektoren betrachtet werden. Die Summe der Produkte kann dadurch wiederum schlicht als *Vektorpunktprodukt* verstanden werden (siehe Gleichung 5).

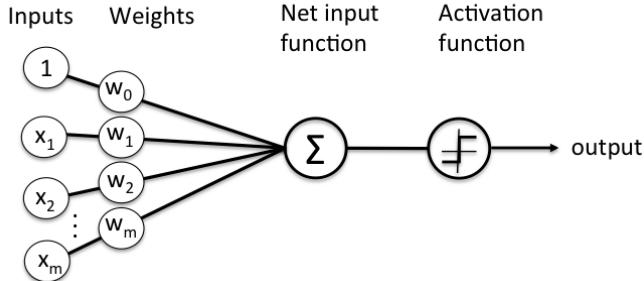
Um die generelle Notation des gesamten Modells zu vereinfachen bietet es sich außerdem an den Schwellwert θ auf die linke Seite der Gleichung (siehe Gleichung 1) zu ziehen. Damit gelten folgende Rahmenbedingungen:

$$g(\mathbf{z}) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{if } z < 0 \end{cases} \quad (4)$$

wobei gilt

$$z = \mathbf{w}_0 \mathbf{x}_0 + w_1 x_1 + \cdots + w_m x_m = \sum_{j=0}^m x_j w_j = \mathbf{w}^T \mathbf{x} \quad (5)$$

Wichtig hierbei, es wird ein zusätzliches Gewicht welches den negativen Schwellwert hält, sowie einen zusätzlichen Inputwert mit dem Wert 1, eingeführt ($w_0 = -\theta$ und $x_0 = 1$). Bei der Berechnung (siehe Gleichung 4) fließt dieser Faktor nun als negativer Summand mit ein wodurch man nur noch prüft ob die Gesamtsumme kleiner Null ist oder nicht. Von dieser Notation wird insbesondere bei der *Lernregel* (siehe Abschnitt 2.2) Gebrauch gemacht. Mittels dieser Vereinfachung lässt sich ein Modell auch folgendermaßen darstellen:



Schematic of Rosenblatt's perceptron.

Abbildung 7: Perceptron - Modelansicht [11]

Letztendlich sei jedoch erwähnt, dass es in der Literatur auch öfters eine Darstellung mittels eines Schwellwerts (engl. *Bias*) gibt, siehe Gleichung 6 [14] ²

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases} \quad (6)$$

²Die Form der Ungleichungen sowie die genauen Ausgabewerte können je nach Quelle variieren.

2.2. Lernregel

Erklärung Das bisher vorgestellte Modell beinhaltet bis jetzt wenig Eigenschaften für einen *lernenden* Algorithmus, dies ändert sich jedoch mit der Idee von Rosenblatt das Modell selbst die Angleichung der Gewichte übernehmen zu lassen. Hierzu wird auf eine Menge von Trainingsdatensätzen zurückgegriffen. Diese Datensätze bestehen aus Eingabewerten für das System und der korrespondierenden Ausgabe. Die Lernregel selbst sieht zusammengefasst folgendermaßen aus.

- Initialisiere die Gewichte mit einem sehr kleinen Wert oder dem Wert 0.
- Für jeden Datensatz der Menge:
 - Berechne den Ausgabewert des Systems
 - Gleiche die Gewichte an

Der genannte Ausgabewert wird über die Aktivierungsfunktion des Modells bestimmt. Die einzelnen Komponenten des *Gewichtsvektors* werden getrennt betrachtet und angeglichen. Ein Aktualisieren eines einzelnen Gewichts innerhalb des Gewichtsvektors wird formal mit $w_j := w_j + \Delta w_j$ beschrieben (das Delta kennzeichnet, dass es sich um eine *Veränderung* handelt). Da die beschriebene Lernregel inkrementell arbeitet, muss man eine sogenannte *Lernrate* bestimmen. Dieser Wert bestimmt letztendlich, wie stark ein Gewicht bei einer Iteration verändert wird. Formal wird diese Rate mit dem Zeichen η dargestellt. Die allgemeine Darstellung zum Angleichen der Gewichte sieht folgendermaßen aus:

$$\Delta w_j = \eta (\text{target}^{(i)} - \text{output}^{(i)}) x_j^{(i)} \quad (7)$$

Zuerst wird ein Datensatz gewählt durch dessen Ausgabewerte alle Gewichte angeglichen werden sollen. Generell gilt, dass das i in den Klammern nicht als Exponent, sondern als Index für den betrachteten Trainingsdatensatz steht. Dann wird die Differenz des optimalen Ausgabewerts und des erreichten Werts berechnet. Diese Differenz wird anschließend mit der Lernrate und dem korrespondierenden Eingabewert des betrachteten Gewichts multipliziert. Dies wird vielleicht etwas klarer, wenn man noch einmal einen Blick in das Diagramm 7 auf Seite 9 wirft. Der Eingabewert $x_j^{(i)}$ steht also für die Komponente mit dem Index j in dem Eingabevektor des Trainingsdatensatzes an der Stelle i innerhalb der Menge von Trainingsdaten.

Anwendung Für einen zweidimensionalen Trainingsdatensatz³ würde die Lernregel folgendermaßen aussehen. Hervorzuheben sei jedoch, dass alle Gewichte gleichzeitig angeglichen werden.

³Ein zweidimensionaler Datenpunkt besitzt einen zweidimensionalen Eingabevektor (also genau zwei Eingabewerte).

$$\begin{aligned}
 \Delta w_0 &= \eta(\text{target}^{(i)} - \text{output}^{(i)}) \\
 \Delta w_1 &= \eta(\text{target}^{(i)} - \text{output}^{(i)}) x_1^{(i)} \\
 \Delta w_2 &= \eta(\text{target}^{(i)} - \text{output}^{(i)}) x_2^{(i)}
 \end{aligned} \tag{8}$$

In dieser Darstellung wurde die bereits in 2.1 besprochene Notation verwendet bei der ein zusätzliches Gewicht mit dem Schwellwert eingeführt wurde. Da der eigentliche Eingabewert hierbei lediglich den Faktor 1 besitzt, fällt er hierbei einfach weg. Wir haben also neben diesem einbezogenen *Bias* lediglich die zwei weitere Gewichte, die angeglichen werden.

Für den Fall, dass das Modell einen Datensatz richtig klassifiziert gibt es genau zwei Möglichkeiten:

$$\begin{aligned}
 \Delta w_j &= \eta((-1^{(i)}) - (-1^{(i)})) x_j^{(i)} = 0 \\
 \Delta w_j &= \eta(1^{(i)} - 1^{(i)}) x_j^{(i)} = 0
 \end{aligned} \tag{9}$$

Durch den Differenzfaktor von Null verändert sich das betrachtete Gewicht erwartungsgemäß nicht. Bei dem entsprechenden Gegenteil ist dies nicht der Fall, hier könnte es zum Beispiel so aussehen:

$$\begin{aligned}
 \Delta w_j &= \eta(1^{(i)} - (-1^{(i)})) x_j^{(i)} = \eta(2) x_j^{(i)} \\
 \Delta w_j &= \eta((-1^{(i)}) - 1^{(i)}) x_j^{(i)} = \eta(-2) x_j^{(i)}
 \end{aligned} \tag{10}$$

Eine Implementierung des beschriebenen Konzepts in der Programmiersprache Python ist auf Github [12] zu finden. Ich werde hier jedoch nicht weiter auf die Details der Implementierung eingehen.

Wie schon die McCulloch-Pitts Zelle (siehe Abbildung 2 Seite 6) ist das Perceptron nur in der Lage mittels einer linearen Klassifizierungs-Funktion die zwei unterschiedlichen Gruppen auseinander zu halten (siehe Abbildung 8).

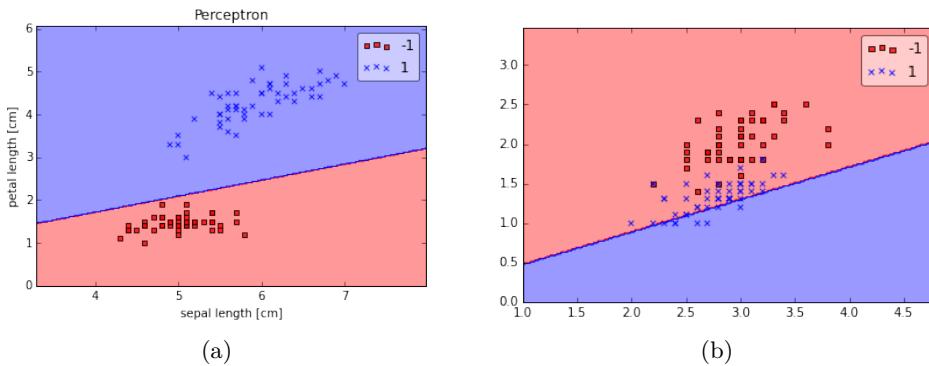


Abbildung 8: Perceptor - Problematische Klassifizierung [11]

3. Adeline

3.1. Aufbau

In 1959 entwickelten der Stanford Professor Bernard Widrow und der Elektroingenieur Marcian Edward Hoff das sogenannte *Adeline-Modell*. Der Name ist ein Akronym für *ADaptive LINEar Element*. Dieses Modell basiert auf dem Perceptron mit dem Unterschied, dass dieses Modell auf die Einheits-Sprungfunktion, wie sie das Perceptron bei der Angleichung der Gewichte verwendet, verzichtet. Es wird stattdessen eine lineare Aktivierungsfunktion $g(z)$ verwendet, welche in diesem Fall erstmal mit der Identitätsfunktion besetzt wird (es gilt also $g(w^T x) = w^T x$). Außerdem wird eine Entscheidungsfunktion an das Ende des ganzen Modells gehängt um weiterhin die Werte quantifizieren zu können. Diese hat jedoch keinen Einfluss auf den Trainingsalgorithmus.

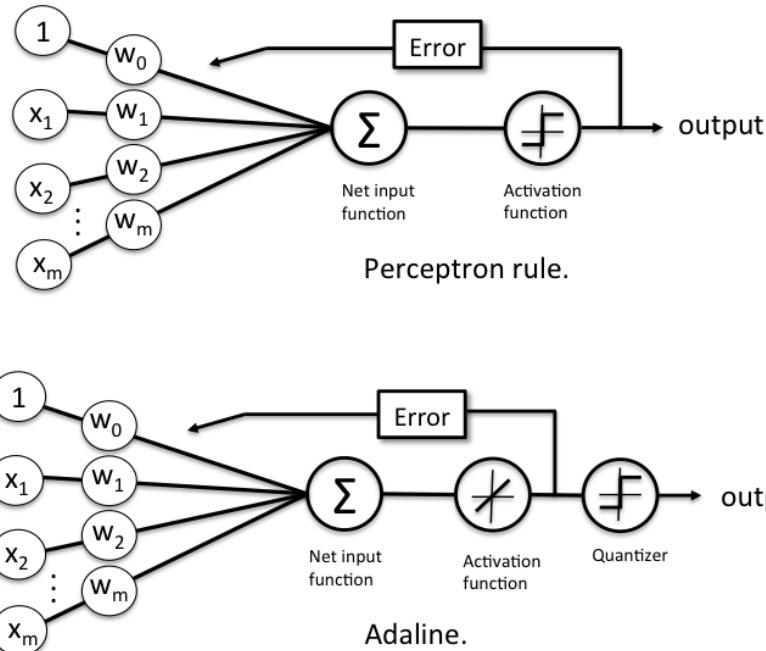


Abbildung 9: Adeline - Aufbau [11]

3.2. Lernalgorithmus

Widrow und Hoff definierten die Delta-Regel für den Lernalgorithmus ihres Modells. Dieser ist auch unter dem Namen *Least-Mean-Square-Algorithmus* bekannt und ist auch heute noch von Relevanz. Im Kern möchte man hierbei das Minimum einer Kostenfunktion über dem Modell bestimmen. Ich werde im folgenden Abschnitt darauf eingehen, wie, dieser funktioniert und wie genau er für dieses Modell eingesetzt werden kann.

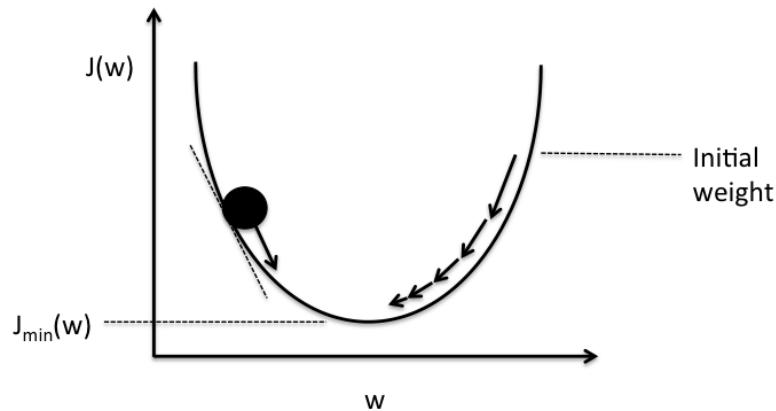
Gradientenverfahren Der wesentliche Nachteil der Einheits-Sprungfunktion ist der, dass sie nicht stetig und damit auch nicht differenzierbar ist. Deswegen hat man sich beim Lernalgorithmus des Adeline-Modells dazu entschieden stattdessen die Identitätsfunktion zu verwenden.

Es wird zuerst eine Kostenfunktion $J(w)$ definiert die minimiert werden soll. Die Kostenfunktion wird durch die *Regressionsquadratsumme*⁴ definiert. Die Formel sieht folgendermaßen aus:

$$J(w) = \frac{1}{2} \sum_i (\text{target}^{(i)} - \text{output}^{(i)})^2 \quad \text{output}^{(i)} \in \mathbb{R} \quad (11)$$

Wichtig hierbei, der Vorfaktor $\frac{1}{2}$ gehört nicht zur herkömmlichen Regressionsquadratsumme, wurde hier jedoch hinzugefügt um später einfacher ableiten zu können. Ziel ist es die bestimmte Abweichung über alle Trainingsdaten so minimal wie möglich zu gestalten. Dazu muss man die Gewichte sowie die Schwellwerte entsprechend anpassen. Um das zu tun, reicht es das Minimum dieser Funktion zu finden. Dazu bedient man sich einer Technik namens *Gradientenverfahren* (englisch *gradient descent*).

Werfen wir einen Blick auf die Abbildung 10.



Schematic of gradient descent.

Abbildung 10: Gradientenverfahren mit eindimensionaler Kostenfunktion [11]

Die dargestellte Funktion besitzt nur einen Eingabewert. Zur Veranschaulichung kann man sich einen Ball vorstellen, der einen Berg bzw. in ein Tal herunter rollt. Bezogen auf das Beispiel betrachtet man einen beliebigen Funktionswert und bestimmt die Ableitung an dieser Stelle. Diese Ableitung kann auch als „Steigung“ an der betrachteten Stelle verstanden werden. Wenn man diese nun invertiert hat, bekommt man theoretisch die „Richtung“ in die der Ball rollen müsste. Ähnlich wie schon der Lernalgorithmus beim Perceptron, wird mit einer Lernrate η gearbeitet, die bestimmt wie viel Veränderung in einem Iterations-Schritt stattfinden soll. Im Beispiel könnte man diese als Schrittweite

⁴engl. sum of squared errors (SSE)

verstehen. Wie weit in einem Schritt gearbeitet wird, ist von fundamentaler Bedeutung. In Abbildung 11 ist zu sehen, dass es problematisch sein kann die Lernrate zu hoch zu definieren, denn der Algorithmus kann das Minimum auch überspringen. Andererseits kann es auch zu Problemen kommen wenn die Lernrate zu klein definiert wurde, da der Algorithmus eventuell in einem lokalen Minimum „steckenbleibt“. Um gerade das erste Problem zu beheben wird die Lernrate heutzutage in jedem Iterationsschritt in Abhängigkeit der Größe der Steigung berechnet.

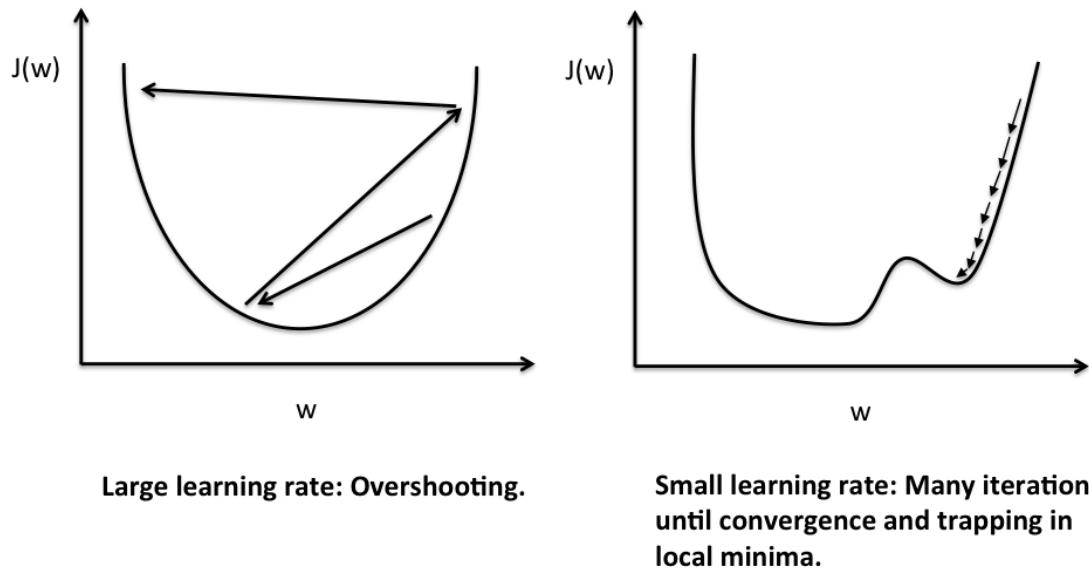


Abbildung 11: Gradientenverfahren - unterschiedliche Lernraten [11]

Letztendlich besitzt das hier betrachtete Modell aber meist mehr als ein einzelnes Gewicht, weswegen man sich nun damit auseinander setzen muss, wie man dieses Verfahren auf eine Kostenfunktion mit einem mehrdimensionalen Vektor anpassen muss. Mit einem zweidimensionalen Eingabevektor ist dies noch relativ gut darstellbar (siehe Abbildung 12).

Auch bei Funktionen mit mehreren Eingabewerten ist es möglich an einem betrachteten Eingabevektor die Steigung zu bestimmen. Allgemein wird diese als *Gradientenvektor* bezeichnet. Bei der gerade betrachteten Abbildung bilden die *x*- und die *y-Achse* jeweils die beiden Eingabewerte der Funktion, wobei die Ausgabe mit der *z-Achse* dargestellt wird. Hier wird die Analogie mit dem Herabrollen eines Berges vielleicht noch etwas klarer. Auch hier gilt, wie schon bei den anderen betrachteten Modellen, wieder die Notation:

$$w = w + \Delta w \quad (12)$$

Δw stellt den angedeuteten Gradientenvektor dar. Im mehrdimensionalen Raum ist

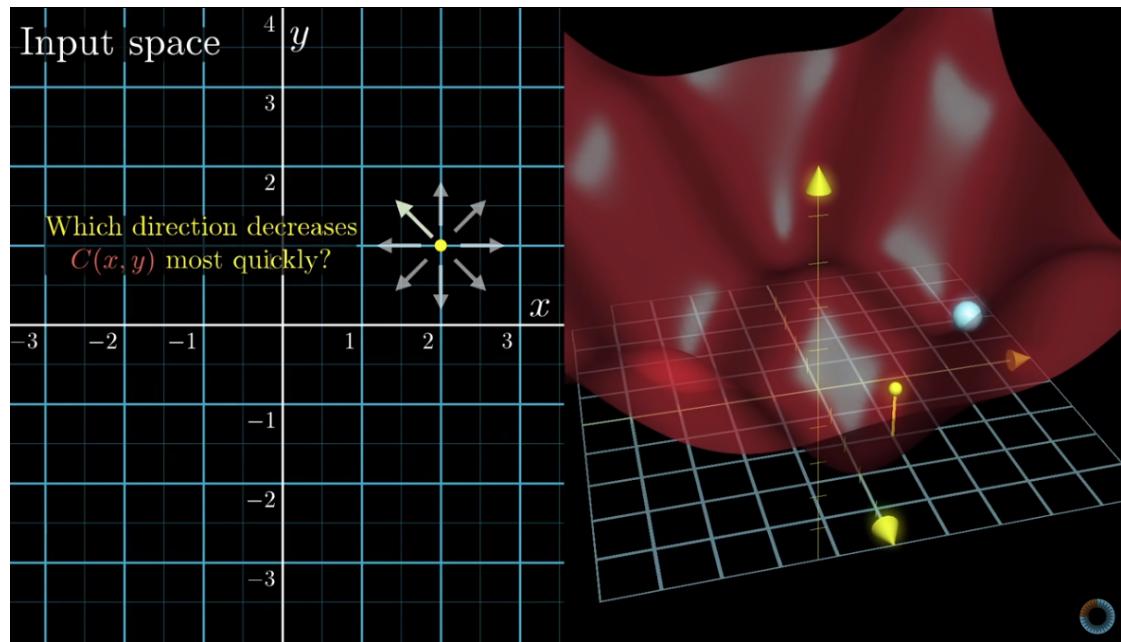


Abbildung 12: Gradientenverfahren - zweidimensionaler Eingabevektor [11]

diese Gewichtsveränderung im allgemeinen Fall den kompletten Gewichtsvektor und für den speziellen Fall mit einem einzelnen Gewicht folgendermaßen definiert.

Regel 3.1: Gradient

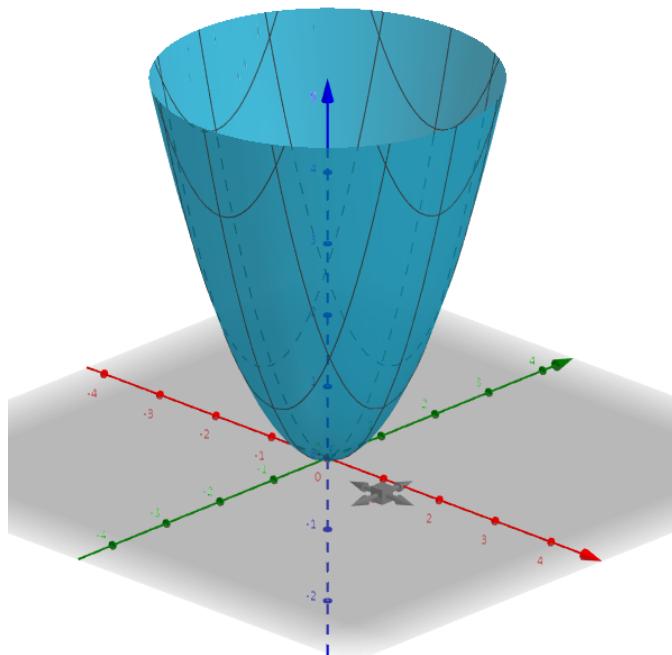
Allgemein:

$$\Delta w = -\eta \nabla J(w) \quad (13)$$

Für die jeweiligen Gewichte:

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} \quad (14)$$

Exkurs - Partielle Ableitungen [9] Da es sich im Beispiel um einen mehrdimensionalen Eingabevektor handelt, muss man mit den partiellen Ableitungen arbeiten. Diese lassen sich am besten anhand eines weiteren Beispiels erklären. Betrachten wir zuerst einmal die Funktion $f(x, y) = x^2 + y^2$ (siehe Abbildung 13).

Abbildung 13: Partielle Ableitung 1⁵

Wann man nun einen Punkt auf der Oberfläche der Funktion betrachtet ist es nicht möglich zu sagen wie stark die Steigung ist, denn es fehlt eine Richtung bezüglich der „geschaut“ werden soll. Hierzu betrachten wir noch einmal die Abbildung 14. Hier wurde eine „rote Ebene“ eingeführt. Dies kann als *Blickwinkel* angesehen werden. Wenn man nun einen Punkt auf dieser Schnittgerade betrachtet ist es möglich die zugehörigen Steigung zu ermitteln. Um die Ebene in der betrachteten Abbildung zu erzeugen wird einfach eine der beiden Eingabeparameter auf einen Wert festgelegt (in diesem Fall $y = 1$). Man kann natürlich auch den Wert für den Eingabeparameter x festlegen und y als Laufvariable betrachten.

Dieser Prozess des Aufteils der Ableitungen in die einzelnen Teile wird als *Partielle Ableitung* bezeichnet. Wir nehmen eine Funktion mit mehrdimensionalem Eingabevektor und berechnen für jede Komponente des Vektors die Ableitung. Kurze Anmerkung zur Notation: Im Zähler des Bruchs welcher die Partielle Ableitung angibt steht stets die Funktion welche abgeleitet werden soll, während man im Nenner die Variable bezüglich der abgeleitet werden soll notiert.

⁵Mit Geogebra 3d erstellt: <https://www.geogebra.org/3d>

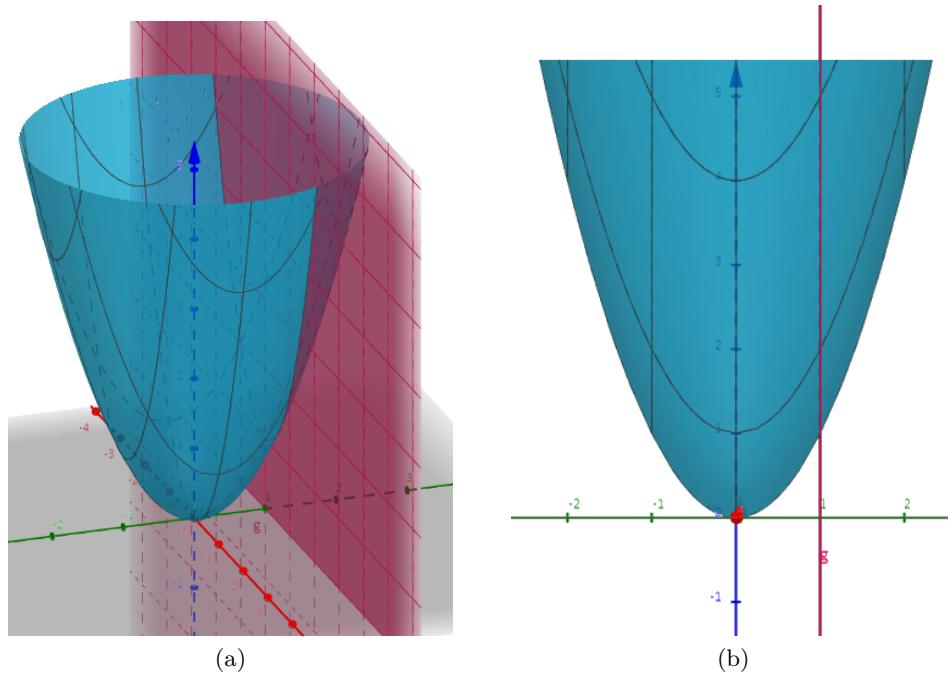


Abbildung 14: Partielle Ableitung 2

In unserem Beispiel besitzt die Funktion $z = f(x, y) = x^2 + y^2$ folgende Ableitung:

$$\begin{aligned} z &= f(x, y) = x^2 + y^2 \\ \frac{\partial z}{\partial x} &= 2x \\ \frac{\partial z}{\partial y} &= 2y \end{aligned} \tag{15}$$

Um nun den Gradienten zu bestimmen muss man diese einfach nur aufzufaddieren:

$$\Delta J \approx \frac{\partial J}{\partial w_1} \Delta w_1 + \frac{\partial J}{\partial w_2} \Delta w_2. \tag{16}$$

Batch-Gradientenabstieg Um den Gradientenvektor der Kostenfunktion (Regressionsquadratsumme) zu bestimmen greift folgender Formelapparat welchen ich im folgenden schrittweise durchgehen werde.

Herleitung 3.1: Gradient

$$\frac{\partial J}{\partial w_j} = \frac{\partial}{\partial w_j} \frac{1}{2} \sum_i (t^{(i)} - o^{(i)})^2 \quad (17)$$

$$\frac{\partial J}{\partial w_j} = \frac{1}{2} \sum_i \frac{\partial}{\partial w_j} (t^{(i)} - o^{(i)})^2 \quad (18)$$

$$\frac{\partial J}{\partial w_j} = \frac{1}{2} \sum_i 2(t^{(i)} - o^{(i)}) \frac{\partial}{\partial w_j} (t^{(i)} - o^{(i)}) \quad (19)$$

$$\frac{\partial J}{\partial w_j} = \sum_i (t^{(i)} - o^{(i)}) \frac{\partial}{\partial w_j} \left(t^{(i)} - \sum_j w_j x_j^{(i)} \right) \quad (20)$$

$$\frac{\partial J}{\partial w_j} = \sum_i (t^{(i)} - o^{(i)}) (-x_j^{(i)}) \quad (21)$$

1. Gleichung 17: Da wir den Gradienten der Kostenfunktion (Regressionsquadratsumme) bilden wollen müssen wir diese bezüglich des gerade betrachteten Gewichts ableiten. Deswegen notiert man im Zähler die Funktion J auf der linken Seite der Gleichung und schreibt sie auf der rechten Seite einfach einmal aus. Im Nenner befindet sich das Gewicht an der Stelle j .
2. Gleichung 18: Die Summe sowie der Faktor $\frac{1}{2}$ kann generell aus der Ableitung rausgezogen werden ohne etwas am Gesamtergebnis zu verändern.
3. Gleichung 19: Ein erster Ableitungsschritt bezüglich der Kettenregel wird angewandt. Die äußere Funktion $2(t^{(i)} - o^{(i)})$ wird vor die noch ausstehende Ableitung der inneren Funktion $\frac{\partial}{\partial w_j} (t^{(i)} - o^{(i)})$ gehängt.
4. Gleichung 20: Der konstante Faktor 2 wird aus der Summe herausgezogen und direkt mit dem am Anfang hinzugefügten Faktor $\frac{1}{2}$ multipliziert. Außerdem wird die eigentliche Berechnung der Ausgabe für $o^{(i)}$ eingesetzt ($\sum_j w_j x_j^{(i)}$).
5. Gleichung 21: Es wird die innere Ableitung nach dem Gewicht w_j gebildet. Das $t^{(i)}$ entfällt, da es sich um einen konstanten Faktor handelt. Der Faktor -1 kann vorgezogen werden womit man nur noch die Summe $\sum_j w_j x_j^{(i)}$ ableiten muss. Wie bereits beim Perceptron (siehe Gleichung 5) bildet sich die letztendliche Ausgabe ja über das Vektorprodukt $w^T x$. Wenn man die Summe ausschreibt fällt einem auf, dass es nur einen einzigen Term in der kompletten Summe gibt welcher den Faktor w_j beinhaltet über welchem hier schließlich abgeleitet werden soll. Daher ist es möglich die gesamte Summe auf $-x_j^{(i)}$ abzuleiten.

Regel 3.2: Gradient

Gradientenkomponente:

$$\frac{\partial J}{\partial w_j} = \sum_i (t^{(i)} - o^{(i)}) (-x_j^{(i)}) \quad (22)$$

Dies muss nun in die ursprüngliche Form (siehe Gleichung 3.2 Seite 15) eingefügt werden:

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = -\eta \sum_i (t^{(i)} - o^{(i)}) (-x_j^{(i)}) = \eta \sum_i (t^{(i)} - o^{(i)}) x_j^{(i)} \quad (23)$$

Der komplette Gradientenvektor beinhaltet die Ergebnisse für die Ableitungsschritte nach jedem einzelnen Gewicht:

$$\nabla J \equiv \left(\frac{\partial J}{\partial w_1}, \dots, \frac{\partial J}{\partial w_m} \right)^T. \quad (24)$$

Die einzelnen Komponenten des Gradientenvektors können dann wieder mit der allererste Formel (siehe Gleichung 12) in neue Gewichte verrechnet werden.

Die wesentlichen Unterschiede zum Lernalgorithmus, welcher beim Perceptron verwendet wird, bestehen daraus, dass der Ausgabewert bei diesem Modell mit einer reellen Zahl dargestellt wird. Außerdem wird für das Aktualisieren eines Gewichts die komplette Menge an Trainingsdatensätzen verwendet. Deswegen wird diese Herangehensweise auch *Batch-Gradientenabstieg* genannt. Beim Perceptron wird das Gewicht nach jedem einzelnen Trainingsdatensatz gerichtet. Eine Implementierung dieses Algorithmus befindet sich ebenfalls in dem Github Repository [12].

Stochastischer Gradientenabstieg Der vorgestellte Algorithmus ist in der Praxis aber relativ rechenaufwendig weil man für jeden einzelnen Iterationsschritt alle Trainingsdatensätze betrachten muss. Ein Weg dies zu umgehen ist der sogenannte *Mini-Batch Gradientenabstieg*. Hierbei wird die komplette Menge an Traininngsdatensätzen erst einmal gemischt und anschließend in kleinere Mengen unterteilt. Beim Iterationsschritt wird dann jeweils eine solcher Mengen betrachtet. Dies ist nicht der effizienteste Weg allerdings ist dieser viel schneller berechnet, dennoch gibt er eine gute Annäherung dafür wie die einzelnen Gewichte geändert werden müssen. In Abbildung 15 kann man erkennen wie die unterschiedlichen Arten des Gradientenabstiegs grob arbeiten. Der linke Graph beschreibt den „herkömmlichen“ Batch-Gradientenabstieg während der rechte den stochastischen darstellt. Dieser trifft zwar nie die genau Richtung ist allerdings immer relativ nah dran und findet letztendlich ebenfalls das Ziel.

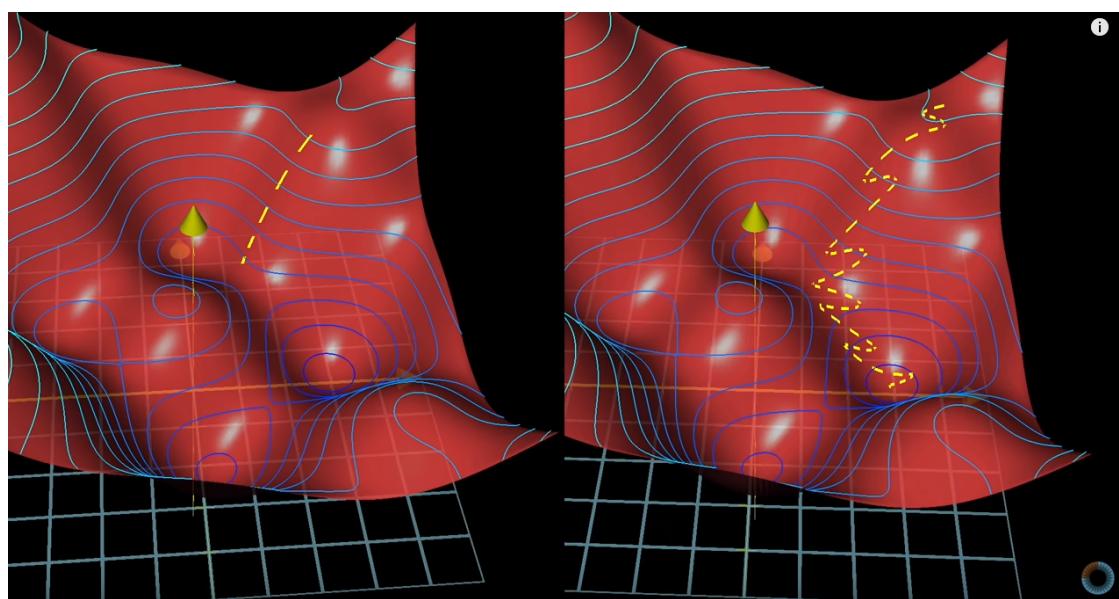


Abbildung 15: Gradientenverfahren - Vergleich [1]

4. Convolutional Neural Network

Bisher wurden in erster Linie die Grundlagen für die heutigen neuronalen Netze erläutert. Im kommenden Abschnitt werde ich grob darauf eingehen, wie man diese Techniken nutzen kann um mithilfe eines *Convolutional Neural Networks* Bilder zu erkennen.

4.1. Geschichte

Zellarten Im Jahr 1959 entdeckten die beiden Neurophysiologen Torsten Wiesel und David H. Hubel die sogenannten *simple* und *complex cells*. Sie beschrieben einen groben Zusammenhang dafür, wie diese beiden Zellarten bei der Mustererkennung im visuellen Kortex verwendet werden.

- Die *simple cells* können einfache Kanten und Balken mit einer bestimmten Orientierung erkennen. Eine solche Zelle könnte zum Beispiel in der Lage sein einen Balken am unteren Bildrand als solchen zu erkennen.
- Eine *complex cell* ist ebenfalls dazu in der Lage diese Formen zu erkennen, allerdings mit dem Zusatz diese Konstellation von Formen auch an verschiedenen Positionen des Bildes zu erkennen. Bezogen auf das Beispiel vom letzten Punkt könnte diese Zellart auch in der Lage sein solche Balken in der Mitte oder am oberen Rand des Bildes zu erkennen. Diese Eigenschaft der Positionsabhängigkeit eines Musters wird *spatial invariance* genannt („räumliche Invarianz“).

Einige Jahre später (1962) beschrieben die beiden Wissenschaftler, wie genau solche komplexen Zellen diese Eigenschaft erreichen. Diese Zellen summieren die Ausgaben von mehreren *simple cells* auf. Alle beteiligten Zellen sind auf die gleichen Formen spezialisiert, analysieren jedoch jeweils einen unterschiedlichen Teil des Bildes (englisch: *receptive fields*, siehe Abbildung 16). So kann zum Beispiel eine komplexe Zelle horizontale Balken an mehreren Positionen im Bild erkennen, indem sie auf die unterschiedlichen Ausgabewerte von mehreren simplen Zellen zurückgreift und diese aufaddiert. Diese Herangehensweise des Herunterbrechens einer komplexen Aufgabe in mehrere einfachere Aufgaben ist ein wesentliches Merkmal aller neuronalen Netze sowie der menschlichen Wahrnehmung im visuellen Kortex.

In den 1980er Jahren entwickelte Dr. Kunihiko Fukushima das Modell von Hubel und Wiesel weiter, indem er ein mathematisches Modell mit den beiden Typen *S-Cells* und *C-Cells* einführte. Die S-Cells befinden sich jeweils in der ersten Schicht des Modells und sind mit den C-Cells verbunden.

Erkennung von Handschrift Einer der Pioniere auf dem Gebiet ist der französische Informatiker Yann LeCun. In den 90er Jahren publizierte er diverse Ausarbeitungen. In seiner bekanntesten beschreibt er wie ein einfaches CNN Modell in der Lage ist, handschriftliche Ziffern zu erkennen. Sein Modell verwendet wie schon angedeutet die Eigenschaft mit einfach Formen komplexere zu bilden und somit die Ziffern zu erkennen. Um sein Modell zu trainieren verwendete er die *MNIST database of handwritten digits*.

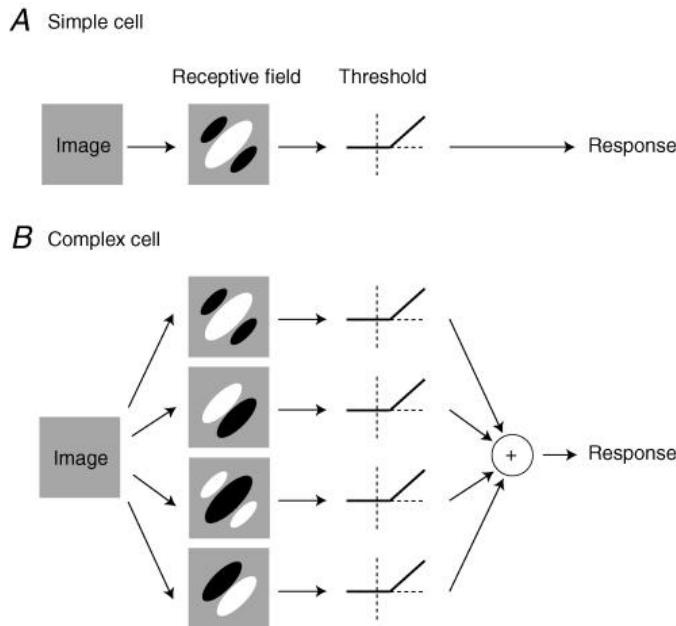


Abbildung 16: Vergleich - Simple und Complex Cell [8]

Diese Datenbank enthält Bilder von handgeschriebenen Ziffern, welche jeweils mit einem entsprechenden Label versehen wurden. Diese Daten werden vom Netz verwendet um die Kostenfunktion aufzustellen⁶. Die Datenbank enthält circa 60.000 Datensätze, welche ausschließlich zum Trainieren des Netzes verwendet werden und 10.000 Datensätze um den letztendlichen Fehler zu berechnen. Wie genau dies geschieht wird im späteren Verlauf des Kapitels geklärt (siehe *Backpropagation* Kapitel 6).

4.2. Funktionsweise

Gegeben sei ein beliebiges Bild, ein CNN soll nun in der Lage sein dazu anzugeben mit welcher Wahrscheinlichkeit das Bild zu einer definierten Klasse gehört. Dies können durchaus auch mehr als eine Klasse pro Bild sein. Das zu kategorisierende Bild wird in Form einer Matrix verarbeitet. In Abbildung 17 kann man grob erkennen, wie so etwas aussehen könnte.

Die rechte Darstellung ist hierbei auch noch unvollständig, denn es handelt sich bei dem Bild um ein Farbbild. In der Praxis werden für jedes Pixel drei unterschiedliche Farbwerte zwischen 0 und 255 gespeichert (wenn man sich im RGB-Farbraum befindet). Um die Ausgabeklasse eines Eingabearrays zu bestimmen wird wie schon bei den vorherigen Modellen darauf gesetzt von *low-level* Eigenschaften, wie bestimmte Formen an bestimmten Positionen, auf *high-level* Eigenschaften geschlossen. Um dies zu erreichen wird bei dieser Art von Netz mit mehreren sogenannten Layern gearbeitet. Das Modell besteht im wesentlichen aus zwei Arten von Schichten: den Filtern (*convolutional layer*)

⁶Es sei ebenfalls noch kurz erwähnt, dass es zu sehr vielen Themengebieten derartige Datenbanken gibt. Eine umfangreiche Liste von frei verfügbaren Quellen ist hier [5] zu finden.



Abbildung 17: Vergleich - Darstellung Mensch und Maschine [4]

und Aggregations-Schichten (*Pooling Layer*). Diese wiederholen sich abwechselnd. In Abbildung 18 ist zu erkennen wie die beschriebenen Zwischenschritte der Verarbeitung grob aussehen können.

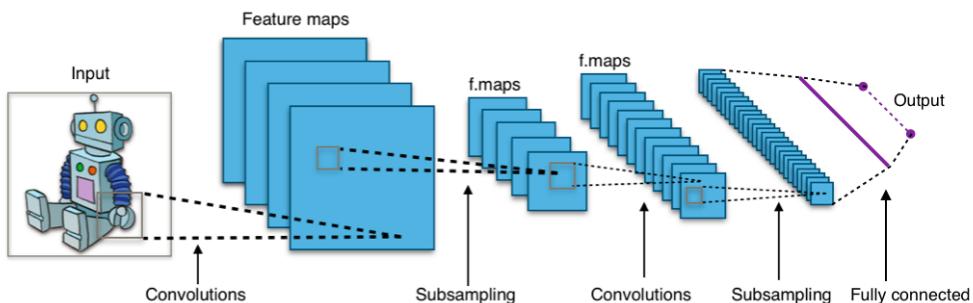


Abbildung 18: Überblick - CNN Verarbeitungsschritte [6]

Convolutional Layer Der gegebene Matrix-Input wird mittels sogenannter *Filter* (auch *Neuron* oder *Kernel* genannt) analysiert. Ein Filter besitzt eine feste Pixelgröße (*Kernel-Size*) wie zum Beispiel 5×5 . Diese spannt ein kleines „Fenster“ über der Matrix auf. Dieses Fenster scannt anschließend mit einer definierten Schrittweite zeilenweise die Eingabematrix. Mittels eines Parameters *Padding* wird festgelegt, wie sich der Filter verhalten soll, wenn er den Rand der Matrix erreicht. Die betrachteten Pixel im Betrachtungsfenster aggregieren zu einem Neuron in der nächsten Schicht. Die Abbildung 19 zeigt wie der sogenannte erste *hidden Layer* dadurch aufgebaut wird. Wichtig: *Die Größe dieser Ergebnismatrix ist abhängig von der Größe (Kernel-Size) des Filters, dem Padding und vor allem von der Schrittweite* [6].

Eine Schrittweite von 2 bei einem Betrachtungsfenster von 2×2 Pixeln führt beispielsweise pro Filter zu einer Halbierung der Größe der Ergebnismatrix im Vergleich zur Input-Matrix. Da in diesem Beispiel immer 4 Pixel gleichzeitig an einem Filter „hängen“ wird die Eingabe in gewisser Weise *gefaltet* (englisch *convolution*).

In der Praxis wird oft ein convolutional Layer mit 32 oder 16 Bit Filtern verwendet.

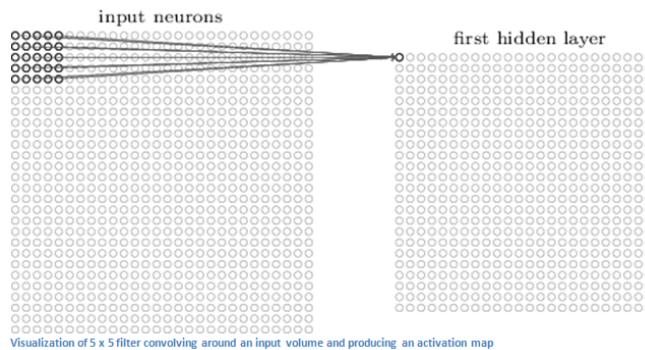


Abbildung 19: CNN - Convolution Layer [6]

Jeder dieser Filter generiert eine eigene Ausgabematrix. Als nächste Schicht folgt erneut ein convolutional Layer, welcher die Ausgabematrizen als neuen Input verwendet. Die dadurch generierte Ausgabe wird auch *activation map* oder *feature map* genannt und wird wiederum in einen *Pooling Layer* gesteckt.

Beispiel - Handschriftliche Ziffern Im folgenden Beispiel betrachten wir die Erkennung einer Ziffer 7 (siehe Abbildung 20). Diese Ziffer besteht aus mehreren Bestandteilen. Es gibt zum Beispiel einen vertikalen Strich von Links unten nach rechts oben, manche schreiben auch noch einen Querstrich in die Mitte der Ziffer. In diesem Einschub wollen wir uns aber dem Querstrich im oberen Teil der 7 widmen.

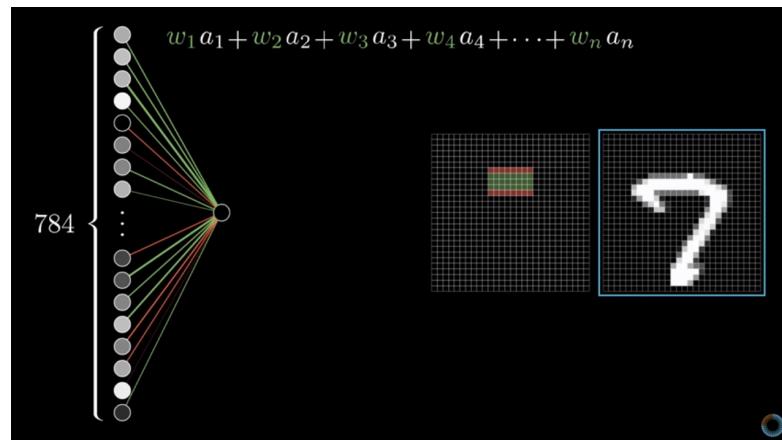


Abbildung 20: CNN - Filter [1]

Um diese Ziffer 7 als eine solche zu erkennen muss das Netz in der Lage sein unter anderem diesen Querstrich zu erkennen. Um diese Eigenschaft zu erkennen bildet man hierfür ein Betrachtungsfenster. Jeder Pixel innerhalb dieses Betrachtungsfensters bekommt ein Gewicht zugewiesen. Die Darstellung ist so zu verstehen, dass die roten Kästen ein negatives Gewicht und die grünen ein positives Gewicht darstellen. Die schwarzen Kästen

sollen dagegen ein Gewicht mit dem Wert 0 enthalten. Wenn man nun die erhaltenen Pixelwerte (in der Abbildung rechts) mit den entsprechenden Gewichten multipliziert und aufaddiert, bekommt man einen recht hohen Wert, wenn in dem gegebenen Bild ein Querstrich in diesem Teil des Bildes vorhanden sein sollte und einen sehr kleinen wenn dies nicht der Fall ist. Durch diesen Sachverhalt ist es möglich eine primitive Eigenschaft des Bildes zu erkennen.

Pro zu erkennendem Merkmal gibt es in einem CNN einen Filter. Jedes Betrachtungsfenster dieses Filters würde dann die erwarteten Farbwerte für die jeweilige Ziffer widerspiegeln. Die Erkenntnisse dieser *low-Level* Filter können dann wiederum mit weiterführenden Filtern verarbeitet werden. Und so können nach und nach auch die komplexesten Formen erkannt werden.

Letztendlich erinnert dieses Prinzip doch sehr stark an die Entdeckung von Hubel und Wiesel. Die einzelnen Neuronen, die innerhalb dieses Netzes verwendet werden, halten sich jedoch stark an die Funktionsweise des Adeline-Modells. Hierbei werden jedoch mehrere Schichten hintereinander gehängt um die Komplexität des Ganzen zu steigern.

Pooling Layer Ein *Pooling Layer* ist dafür zuständig die Ergebnisse von Convolutional Layern zu aggregieren. Dies wird anhand eines Pooling-Prozesses ausgewertet. Der wesentliche Zweck dieser Schicht ist es *nur die relevantesten Signale an die nächsten Schichten weiter zu geben, eine abstraktere Repräsentation des Inhalts zu erreichen und die Anzahl der Parameter eines Netzes zu reduzieren* [6]. Generell gilt, während die Größe des Inputs durch die Faltungen und das Pooling immer weiter reduziert wird, erhöht sich die Anzahl der Filter zur Erkennung von übergeordneten Signalen zunehmend.

Ein weit verbreiteter Pooling-Mechanismus stellt der sogenannte *MaxPooling Layer* dar. Hierbei wird einfach der höchste Wert einer Kernel-Matrix verwendet und alle anderen werden verworfen. Es gibt allerdings auch andere Algorithmen, auf die ich hier jedoch nicht näher eingehen werde:

- fractional max pooling
- L_p pooling
- mean pooling
- stoachastic pooling
- spatial pooling
- generalized pooling

Fully Connected Layer Diese Schicht wird auch oft als *Dense Layer* bezeichnet. Ausgangspunkt für die Verwendung dieses Layers ist, dass sämtliche high-level Merkmale bereits durch die vorangegangenen Schichten erkannt wurden. In dieser Schicht werden sämtliche Neuronen mit allen Ausgabewerten verbunden. Generell fasst diese Schicht alle high-level Merkmale zu den jeweiligen Ausgabeklassen zusammen. Wenn wir beim Beispiel der Ziffernerkennung bleiben, spiegelt die Vorgängerschicht die einzelnen Schnörkel und Striche der einzelnen Ziffern wieder. Der Dense Layer fasst diese dann zu den wirklichen Ziffern zusammen. So entsteht als Ausgabe hierbei ein N-dimensionaler Vektor, welcher in den einzelnen Komponenten die jeweilige Wahrscheinlichkeit hält, dass es sich um die jeweilige Klasse handelt oder nicht. Bei den Ziffern wäre dies ein 10 dimensionaler Vektor. Wenn man nun die Ziffer 0 erkennen wollen würde, kann es gut sein, dass sich

das Netz „nicht ganz sicher“ ist, da die Ziffer 9 im oberen Bereich zum Beispiel ebenfalls eine Rundung besitzt. Die Ausgabe könnte dann so aussehen, dass alle Komponenten bis auf die am Index 0 und 9 mit einer Wahrscheinlichkeit von 0 belegt wurden.

Training Um sämtliche Gewichte und Schwellwerte des Netzes richtig anpassen zu können, wird auf einen Trainingsalgorithmus zurückgegriffen. Das in diesem Fall genutzte Verfahren der *Backpropagation* ist von fundamentaler Bedeutung für alle aktuellen Architekturen, daher bekommt er eine eigenen Abschnitt (siehe Kapitel 6).

5. Fazit

5.1. Beantwortung der Leitfragen

- Woher kommt Deep Learning und wie ist dieser Begriff im Kontext zur künstlichen Intelligenz einzuordnen?
- Welche Entwicklungen hat das Neuronale Netz von damals zu heute durchgemacht?

Bei dem Begriff des Deeplearnings handelt es sich um eine Weiterentwicklung des *Machine Learnings*. Er beschreibt eine Gruppe von Netzstrukturen, welche zahlreiche Zwischenschichten beinhalten (englisch *hidden Layers*). Das *Convolutional Neural Network* ist dabei ein Vertreter einer solchen Struktur. Diese Art von Architektur benötigt spezielle Trainingsmechanismen. Einen werde ich im folgenden Abschnitt näher behandeln (siehe Kapitel 6).

Generell wurden jedoch die wesentlichen Vorstufen grob angerissen. Wir haben die Ursprünge der ersten künstlichen Neuronen (siehe MPN Abschnitt 1, Perceptron Abschnitt 2) gesehen und wie diese auf der Biologie des Menschen basieren. Darüber hinaus wurden die Verbesserungen dieser ersten Modelle vorgestellt (siehe Adeline Abschnitt 3). Außerdem haben ich die ersten Anfänge des künstlichen Lernens mithilfe von Lernalgorithmen sowie die Mathematik dahinter beschrieben (siehe Abschnitt 3.2). Um auch einen kleinen praktischen Einblick zu haben wurde ebenfalls grob beschrieben wie ein Rechner in der Lage ist bestimmte Merkmale eines Bildes zu erkennen (siehe CNN Abschnitt 4).

Teil II.

Aktuelle Entwicklung

Im folgenden Abschnitt werde ich auf die aktuelle Entwicklung der neuronalen Netze eingehen. Insbesondere wird dabei auch noch einmal der Lernalgorithmus (Backpropagation) dieser Netze beschrieben und wofür welche genauen Arten von Architekturen verwendet werden. Die Mathematik hierbei ist durchaus relativ fortgeschritten, sodass sich ein Blick in das frei (kostenlos) verfügbare E-Book *Neural Networks and Deep Learning* von Michael Nielsen [14] sehr lohnt. Daraus resultieren folgende Leitfragen:

- Wo können diese Verfahren eingesetzt werden, bzw. wo werden diese bereits eingesetzt?
- Welche Ansätze und Architekturen sind zur Zeit state-of-the-art?

6. Backpropagation

6.1. Notation

Diesem Lernalgorithmus liegt eine Kostenfunktion zugrunde. Letztendlich wird analysiert, wie sich das Netz momentan für einen Trainingsdatensatz verhält. Wenn das Netz diesen richtig erkennt, ist der ausgegebene Fehler gering, ansonsten ist er hoch.

Um diesen Fehler in einem mehrschichtigen Netz effizient zu minimieren, wird in vielen Fällen der *Backpropagation*-Algorithmus verwendet. Er wurde bereits in den Siebzigerjahren definiert, erlangt jedoch erst im Jahr 1986 mit dem Paper *Learning representations by back-propagating errors* von Rumelhart, Hinton und Williams Bekanntheit.

Im vorherigen Teil habe ich bereits beschrieben, was man unter dem Gradientenabstieg versteht und wie dieser auch bei mehrdimensionalen Funktionen (wie zum Beispiel der hierbei betrachteten Kostenfunktion) verwendet werden kann. Es wurde jedoch noch nicht vorgestellt, wie man auf ein mehrschichtiges Netz bezogen diesen Gradienten überhaupt berechnen und nutzen kann um die Fehler der Trainingsbeispiele zu minimieren.

Gemeinhin wird die Notation, wie sie in Abbildung 21 zu sehen ist, für ein Gewicht verwendet. In Abbildung 22 steht das b_j^l für den Schwellwert (*Bias*) am Neuron mit dem Index j im Layer mit dem Index l . Das Selbe gilt für die Aktivierung welche mit einem a gekennzeichnet wird. Mit den gegebenen Notationen können wir folgende Gleichung für die Aktivierung eines Neurons aufstellen (siehe Gleichung 25).

$$a_j^l = \sigma \left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l \right), \quad (25)$$

Diese Formel sollte bereits aus den vorherigen Kapiteln bekannt sein. Hierbei werden die Gewichte mit den entsprechenden Aktivierungen multipliziert und anschließend aufaddiert. Die generierte Ausgabe wird mit dem Schwellwert verrechnet und in eine Aktivierungsfunktion (z.B. die Sigmoid-Funktion) gesteckt.

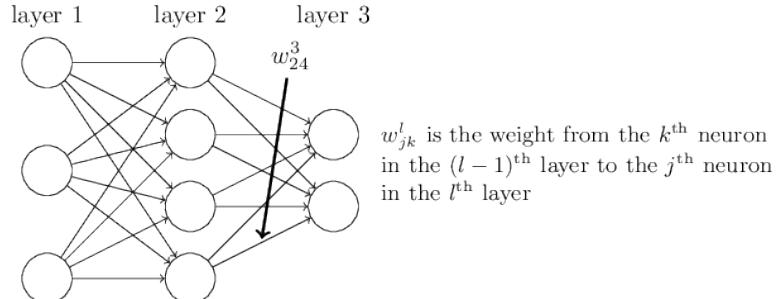


Abbildung 21: Notation [14]

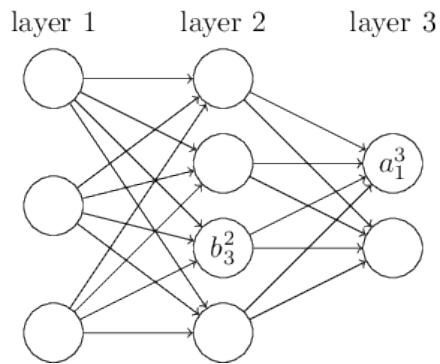


Abbildung 22: Notation [14]

Um später einfacher mit all diesen Werten zu rechnen wird versucht die gegebenen Werte in eine Matrix- beziehungsweise Vektor-Darstellungsform zu bringen. Da Neuronen mehrere ausgehende „Pfade“ besitzen, wird hierbei eine Matrix geformt. Das bisher beschriebene Gewicht w_{jk}^l befindet sich hierbei im Zeilenindex j und Spaltenindex k . Da sich sowohl der Schwellwert als auch die Aktivierung ausschließlich auf ein einzelnes Neuron beziehen, muss hierfür lediglich ein eindimensionaler Vektor pro Layer l generiert werden. Die einzelnen Komponenten werden hierbei über den Index j angesteuert. Die Notation für den Aktivierungsvektor der Schicht l sieht dann derartig aus: a_j^l . Ähnliches gilt für die Schwellwerte b_j^l .

Um mit diesen Vektoren arbeiten zu können muss man die abschließende Aktivierungsfunktion σ vektorisieren⁷. Dabei muss man die Funktion lediglich derartig umschreiben, dass diese auf die einzelnen Komponenten angewendet wird und nicht auf einen einzelnen Wert. Beispielsweise würde die Funktion $f(x) = x^2$ vektorisiert folgendermaßen aussehen:

$$f \left(\begin{bmatrix} 2 \\ 3 \end{bmatrix} \right) = \begin{bmatrix} f(2) \\ f(3) \end{bmatrix} = \begin{bmatrix} 4 \\ 9 \end{bmatrix}, \quad (26)$$

Mit diesen Zusammenfassungen kann nun die Gleichung 25 folgendermaßen umge-

⁷Klarer Verweis auf Nielsen Buch, Kapitel über Backpropagation im Detail [14]

schrieben werden:

$$a^l = \sigma(w^l a^{l-1} + b^l). \quad (27)$$

Diese Umschreibung abstrahiert das Denken über die lokalen Neuronen auf ein höheres Level, sodass es einfacher ist das Gesamtbild zu betrachten. Um noch mehr Klarheit zu schaffen wird oftmals die Aktivierung einer Schicht l aus der Funktion herausgezogen und mit dem Buchstaben z^l versehen. z^l kann nun in die Aktivierungsfunktion σ eingesetzt werden. Im folgenden werde ich diese Ausgabewerte, welche in die Aktivierungsfunktion hereingereicht werden, stets mit z -Wert abkürzen.

Zusammenfassung der bisherigen Beschreibungen.

Regel 6.1: Backpropagation - Notation

Bisherige Schreibweise: Komponentenweise Darstellung

$$a_j^l = \sigma \left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l \right)$$

Zusammengefasste Form: Vektorisierte Form

$$a^l = \sigma(z^l) \quad (28)$$

Gewichtete Eingabe (Z -Werte):

$$z^l \equiv w^l a^{l-1} + b^l \quad (29)$$

6.2. Fundamentale Gleichungen

Ziel des Backpropagation-Algorithmus ist es herauszubekommen welche Gewichte und Schwellwerte verändert werden müssen um die Kostenfunktion zu minimieren. Im folgenden werde ich nach und nach die vier wichtigsten Formeln des Verfahrens beschreiben.

Regel 6.2: Backpropagation - Fundamentale Gleichungen

$$\delta^L = \nabla_a C \odot \sigma'(z^L). \quad (30)$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l), \quad (31)$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l. \quad (32)$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l. \quad (33)$$

Fehler auf der Ausgabeschicht - Gleichung 30 Um zu verstehen was man unter einem Fehler genau versteht sei angenommen, die Ausgabe eines Neurons j im Layer l wird um einen unbestimmten Wert verzerrt. Mathematisch ausgedrückt sieht dies dann folgendermaßen aus:

$$\sigma(z_j^l + \Delta z_j^l) \quad (34)$$

Statt der herkömmlichen Ausgabe $\sigma(z_j^l)$ wird ein Fehler Δz_j^l hinzugefügt. Allgemein wird der Fehler eines einzelnen Neurons dadurch folgendermaßen beschrieben:

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}. \quad (35)$$

Fangen wir jedoch von vorne an. Bei der Backpropagation wird versucht von den Fehlern, welche am Ende auftreten auf die verschiedenen Gewichte und Schwellwerte zu schließen, die verändert werden müssen. Dabei wird zuallererst der durchschnittliche Fehler der letzten Schicht berechnet. Hierfür wird die Kostenfunktion nach dem jeweiligen z-Wert eines Neurons mit Index j abgeleitet.

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} \quad (36)$$

Das L steht diesbezüglich für den letzten (Ausgabe-) Layer. Dieser Ausdruck kann etwas verfeinert werden, indem man die Kettenregel anwendet⁸. Zur Erinnerung, die Kettenregel sieht folgendermaßen aus:

$$\frac{d}{dx} [f(u)] = \frac{d}{du} [f(u)] \frac{du}{dx} \quad (37)$$

Für die Vereinfachung der Gleichung 36 wird die Definition von z herangezogen. Wir erinnern uns, $a^l = \sigma(z^l)$ (siehe Gleichung 28). z stellt in diesem Fall den kompletten Ausgabewert des Neurons dar, *bevor* dieser in die Aktivierungsfunktion gesteckt wurde und berechnet sich wie schon in Gleichung 29 gezeigt mit $z^l \equiv w^l a^{l-1} + b^l$. Wenn man die Kostenfunktion nun nach z ableiten möchte, muss man hierbei erst einmal die äußere mit der inneren Ableitung multiplizieren. Es entsteht folgende Gleichung:

$$\delta_j^L = \sum_k \frac{\partial C}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_j^L} \quad (38)$$

Da die Aktivierung a_k^L im letzten Term lediglich vom Neuron mit dem Index k abhängt, bilden alle Summanden mit $k \neq j$ den Wert 0. Es kann folglich auf die nachfolgende Gleichung geschlossen werden. (Dieses Entfallen der Summe war bereits in der Herleitung des Gradientenabstiegs zu beobachten).

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} \quad (39)$$

⁸Der Youtuber 3blue1brown hat hierzu ein sehr gutes Video produziert, welches die Zusammenhänge mit der Kettenregel verdeutlicht [1] (Kapitel: Backpropagation)

Da folglich gilt, dass $a_j^L = \sigma(z_j^L)$ lässt sich der zweite Term in $\sigma'(z_j^L)$ umschreiben:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L) \quad (40)$$

Die Formel bezieht sich momentan auf eine spezifische Komponente innerhalb der letzten Schicht, Ziel dieser Einheit ist es jedoch möglichst allgemeingültige Formeln zu finden, damit der Rechner effizienter arbeiten kann. Daher wird die betrachtete Formel derartig umformuliert, dass es einen Ausgabevektor gibt, welcher sämtliche Fehler der Ausgabeneuronen enthält. Auf die jeweiligen Felder kann hierbei über die jeweiligen Indizes zugegriffen werden.

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \quad (41)$$

Dies ist auch die Darstellung, wie sie Anfang des Abschnitts verwendet wurde (siehe Gleichungen 6.2). $\nabla_a C$ stellt dabei den Vektor dar, dessen partielle Ableitungen $\partial C / \partial a_j^L$ entsprechen (j ist hierbei weiterhin der Index des betrachteten Neurons innerhalb des letzten Layers). Es ist also lediglich eine andere Schreibweise für die partielle Ableitung, bezogen auf die komplette Ausgabeschicht.

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L) \quad (42)$$

Wenn wir für die Kostenfunktion wie schon in früheren Kapiteln die quadratische Kostenfunktion wählen, entspricht $C = \frac{1}{2} \sum_j (y_j - a_j^L)^2$. Nach a_j^L abgeleitet ergibt sich dann wiederum $\partial C / \partial a_j^L = (a_j^L - y_j)$. Die Herleitung ist sehr ähnlich zu der des Gradienten (siehe Gleichung 6.2), hier fehlt lediglich die Iteration über die kompletten Trainingsdatensätze. Diese Ableitung kann man dann ebenfalls in die Gleichung einsetzen um ein etwas klareres Bild zu schaffen.

$$\delta^L = (a^L - y) \odot \sigma'(z^L) \quad (43)$$

Regel 6.3: Zusammenfassung

Um den Fehlervektor der letzten Schicht zu bestimmen:

$$\delta^L = \nabla_a C \odot \sigma'(z^L). \quad (44)$$

äquivalent zu:

$$\delta^L = (a^L - y) \odot \sigma'(z^L) \quad (45)$$

Um die Fehler komponentenweise zu bestimmen:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L) \quad (46)$$

Rekursive Fehlererkennung - Gleichung 31 Da wir nun die Fehler der letzten Schicht kennen, wird diese Erkenntnis genutzt um auf die Fehler der vorherigen Schichten zu schließen. Im folgenden wird zwischen den beiden Gleichungen $\delta_j^l = \partial C / \partial z_j^l$ und $\delta_k^{l+1} = \partial C / \partial z_k^{l+1}$ ein Zusammenhang hergestellt, der diese Vorgehensweise ermöglicht.

Herleitung 6.1: Rekursiver Fehler

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} \quad (47)$$

$$\delta_j^l = \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} \quad (48)$$

$$\delta_j^l = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1}, \quad (49)$$

$$z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1} \sigma(z_j^l) + b_k^{l+1}. \quad (50)$$

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l). \quad (51)$$

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l). \quad (52)$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l), \quad (53)$$

1. Gleichung 47: Genau wie bereits zur Bestimmung des Fehlers auf der letzten Schicht wird hierbei die Kostenfunktion nach der jeweiligen Ausgabe des Neurons abgeleitet bevor die Ausgabe durch die Aktivierungsfunktion gezogen wird (z_j^l).
2. Gleichung 48: Nun wird wie vorher schon die Kettenregel angewandt (die Aktivierung wird jedoch erst einmal vernachlässigt). Es wird lediglich nach dem Z-Wert aufgelöst.
3. Gleichung 49: Die Reihenfolge der beiden Terme wird vertauscht und die oben genannte Definition eingesetzt ($\delta_k^{l+1} = \partial C / \partial z_k^{l+1}$).
4. Gleichung 50: In einem Zwischenschritt wird der Z-Wert des nächsten Layers notiert. Diese Gleichung ist folgendermaßen zu verstehen: Der Z-Wert des jeweils nächsten Layers im Neuron mit dem Index k setzt sich zusammen aus der Summe aller Gewichte, welche ihren Endpunkt im betreffenden Neuron haben mit den Aktivierungen der vorherigen Schichten. Diese Summe wird anschließend mit dem entsprechenden Schwellwert des Neurons verrechnet. Die Aktivierung a wird abschließend noch durch ihre Funktion mit der Aktivierungsfunktion σ ersetzt.

5. Gleichung 51: Hier wird der gerade besprochene Zwischenschritt, welcher in der Gleichung 49 den vorderen Teil darstellt einmal entsprechend der Notation differenziert. Auch hier gilt wieder, sämtliche Summanden der Summenfunktion mit einem Index $i \neq k$ werden vernachlässigt, da sie bei der Ableitung als Konstanten verstanden werden (es wird schließlich nach z_j^l abgeleitet).
6. Gleichung 52: Wenn wir die letzte Gleichung 51 wieder in unsere ursprüngliche Gleichung 49 einsetzen, bekommen wir diese Darstellung.
7. Gleichung 53: Wie schon bei der letzten fundamentalen Gleichung dieses Algorithmus wird die Darstellung noch einmal von der bisherigen komponentenweisen Darstellung in die vektorielle Darstellung überführt. Wir bekommen eine der fundamentalen Gleichungen vom Anfang dieses Abschnitts.

Schwellwert - Gleichung 32 Mithilfe der Kettenregel wird der entstandene Term derartig umgeformt, dass bereits bestehende Definitionen hier eingesetzt werden können.

Herleitung 6.2: Fehler - Schwellwert

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} \quad (54)$$

$$z_k^l = \sum_j w_{kj}^l a_j^{l-1} + b_k^l = \sum_j w_{kj}^l \sigma(z_j^{l-1}) + b_k^l. \quad (55)$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l. \quad (56)$$

1. Gleichung 54: Mithilfe der Kettenregel können die Derivate wieder auseinandergezogen werden.
2. Gleichung 55: Der *Z-Wert* eines einzelnen Neurons setzt sich aus den Aktivierungen (der vorherigen Schicht), welche in dem beschriebenen Neuron enden und deren Gewichten zusammen. Anschließend wird noch ein Schwellwert aufaddiert. Wenn man diese gesamte Gleichung nach dem Schwellwert ableitet entfällt der vordere Teil komplett, da es sich im Zuge dieser Ableitung dabei um eine Konstante handelt. Der hintere Summand bildet ein Derivat gleich eins.
3. Gleichung 56: Die vorher bestimmte Ableitung wird wieder in die Gleichung 54 eingesetzt. Anschließend wird der hinterbliebene Teil noch durch die Definition, wie sie in Gleichung 47 zu sehen ist ersetzt.

Gewicht - Gleichung 33 Diese Ableitung ist so gut wie identisch zu der im letzten Paragraphen. Dieselbe Funktion z wird hierbei lediglich nach einem anderen Faktor (hier einem gegebenen Gewicht) abgeleitet.

Herleitung 6.3: Fehler - Gewicht

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} \quad (57)$$

$$z_k^l = \sum_j w_{kj}^l a_j^{l-1} + b_k^l = \sum_j w_{kj}^l \sigma(z_j^{l-1}) + b_k^l. \quad (58)$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l. \quad (59)$$

1. Gleichung 57: Genau wie bei der vorherigen Gleichung wird auch hier wieder die Kettenregel angewendet.
2. Gleichung 58: In einem Zwischenschritt wird die Funktion z aufgelistet, nach der im nächsten Schritt abgeleitet werden soll.
3. Gleichung 59: Die im vorherigen Schritt gezeigte Funktion wird nach dem Gewicht abgeleitet. Vom vorderen Summanden bleibt dabei lediglich der Faktor a_j^{l-1} übrig, der hintere Teil entfällt dabei komplett, weil dieser hierbei als Konstante angesehen wird.

6.3. Anwendung

Zuerst werde ich erläutern, wie man den Gradienten der Kostenfunktion bestimmen kann. Anschließend wird der Algorithmus erklärt, der diesen verwendet.

1. **Input x :** Ein Eingabevektor a^l wird in das Netz eingespeist.
2. **Feedforward:** Für jede Schicht $l = 2, 3, \dots, L$ wird der Z-Wert-Vektor mittels $z^l = w^l a^{l-1} + b^l$ und Aktivierungsvektor mit $a^l = \sigma(z^l)$ generiert.
3. **Output Fehler δ^L :** Es wird zuerst der Fehlervektor der letzten Schicht generiert. Dies geschieht mit der ersten beschriebenen Gleichung 30 $\delta^L = \nabla_a C \odot \sigma'(z^L)$.
4. **Output:** Der Gradient der Kostenfunktion wird mithilfe der letzten beiden Gleichungen $\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l$ und $\frac{\partial C}{\partial b_j^l} = \delta_j^l$ generiert.

Der beschriebene Algorithmus ist in der Lage den Gradienten eines einzelnen Trainingsbeispiels zu generieren ($C = C_x$). In der Praxis wird allerdings auf eine Weiterführung wie dem *stochastischen* Gradientenabstieg zurückgegriffen. Hierbei wird die gesamte Menge an Trainingsbeispielen in kleinere Einheiten der Größe m eingeteilt. Wieso dies sinnvoll ist, habe ich bereits im Abschnitt 3.2 erläutert. Im folgenden möchte ich jedoch kurz zusammenfassen, wie dieser notiert wird.

1. Eine Menge an Trainingsbeispielen wird durch das Netz bearbeitet.
2. Für jedes Trainingsbeispiel x : Setze den entsprechenden Aktivierungsvektor $a^{x,1}$ und führe folgende Schritte durch (leicht abgewandelte Version des vorherigen Algorithmus):
 - a) **Feedforward:** Für jede Schicht $l = 2, 3, \dots, L$ wird der Z-Wert-Vektor mittels $z^{x,l} = w^l a^{l-1} + b^l$ und Aktivierungsvektor mit $a^{x,l} = \sigma(z^l)$ generiert.
 - b) **Ausgabe Fehler** $\delta^{x,L}$: Es wird zuerst der Fehlervektor der letzten Schicht generiert. Dies geschieht mit der ersten beschriebenen Gleichung 30 $\delta^L = \nabla_a C \odot \sigma'(z^L)$.
 - c) **Backpropagation Fehler:** Für jede Schicht $l = L-1, L-2, \dots, 2$ wird der entsprechende Fehlervektor mithilfe von $\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l})$ berechnet.
3. **Gradientenabstieg:** Für jede Schicht $l = L, L-1, \dots, 2$ werden die Gewichte entsprechend der Regel $w^l \rightarrow w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$ aktualisiert. Mittels dieser Regel werden alle Gewichtsvektoren Schicht für Schicht durchgegangen. Im hinteren Teil wird der durchschnittliche Fehler über alle m Trainingsbeispiele berechnet, indem man die einzelnen Trainingsbeispiele durchiteriert und für die gerade betrachtete Schicht den Fehlervektor berechnet. Die einzelnen Fehler des Vektors werden dann über die Trainingsbeispiele pro Vektorposition aufaddiert und abschließend durch die Anzahl m geteilt um den Durchschnitt zu erhalten. Im letzten Schritt werden die Durchschnitte noch mit einer Lernrate multipliziert (siehe Abschnitt 2.2).
Um die Schwellwerte einer Ebene zu aktualisieren wird die folgende Regel verwendet: $b^l \rightarrow b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$.

Es sei ebenfalls noch erwähnt, dass man um diesen ganzen Algorithmus durchführen zu können noch eine Möglichkeit benötigt die ganzen Trainingsbeispiele zu mischen und in kleine Batches einzuteilen⁹.

⁹In Nielsens Buch [14] befinden sich zahlreiche Code-Beispiele für die Implementierung von Netzen und ebenfalls eine detaillierte Beschreibung dieses Algorithmus

7. Multilayer Perceptron

Eines der bekanntesten Architekturen der neuronalen Netze ist das sogenannte *Multilayer Perceptron*. Diese sehr vielfältig eingesetzte Struktur verwendet den erläuterten Backpropagation-Lernalgorithmus und erzielt damit in der Regel sehr gute Ergebnisse. Beim MLP spielt die Aktivierungsfunktion durchaus auch eine tragende Rolle und ist nicht fest definiert (häufig wird jedoch die Sigmoid-Funktion verwendet). Wenn das Netzwerk sehr viele verborgene Zwischenschichten besitzt, wird es allgemein als *tief* bezeichnet und es kann zu Problemen beim Training kommen. Hierbei gibt es spezielle Techniken die diese Probleme lösen können und generell unter dem Namen *deep learning* zusammengefasst werden. Da diese Architektur als so vielfältig gilt, kann sie in vielen verschiedenen Anwendungsbereichen eingesetzt werden. Hier ein paar Beispiele:

- Mustererkennung
- Funktionenapproximation
- Klassifizierung
- Prognose
- Diagnose
- Steuerung
- Optimierung

Zusammenfassung - Aufbau Da diese Architektur als Grundlage für die Erklärung des Backpropagation-Algorithmus gewählt wurde, folgt nur eine kurze Zusammenfassung des Aufbaus. Für eine ausführliche Erklärung sämtlicher Notations-Regeln etc. siehe Abschnitt 6.1.

Diese Struktur gilt als mehrschichtige vorwärtsgekoppeltes Netzwerk. Teilweise wird sie auch als Mehrschichten-Perzeptron bezeichnet, weil sie einem Perzeptron-Netzwerk mit mehreren Schichten sehr ähnelt. Dieses Netzwerk besteht aus einer Eingabeschicht, einer oder mehrerer innerer Schichten¹⁰ und einer Ausgabeschicht.

Das Netz kann n Eingabewerte annehmen, welche jeweils mit den Neuronen x_1 bis x_n entgegengenommen werden. Die einzelnen Neuronen werden mit einer Aktivierungsfunktion belegt und verwenden intern einen eigenen Schwellwert. Das durch-Iterieren diverser Eingabewerte wird auch als *Feedforward* bezeichnet, wobei am Ende m Ausgabewerte entstehen, die über die Ausgabeneutronen der letzten Schicht weitergeleitet werden (siehe z_1 bis z_n).

¹⁰in englisch auch häufig *hidden Layers* genannt

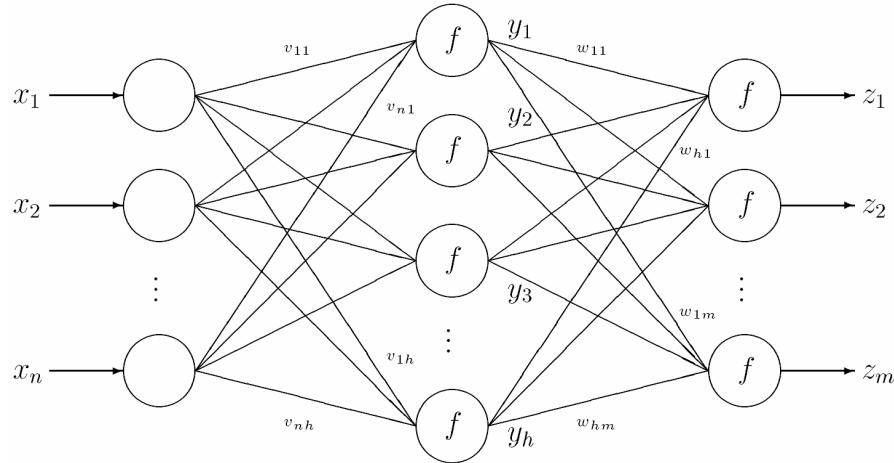


Abbildung 23: MLP - Aufbau [10]

Aktivierungsfunktionen

Sigmoid Um beim Backpropagation-Algorithmus die ersten Ableitungen der Aktivierungsfunktion berechnen zu können, müssen diese stetig differenzierbar sein. Eine häufig verwendete Funktion ist die *Sigmoid*-Funktion. Diese logistische Funktion zeichnet sich dadurch aus, dass alle möglichen Eingabewerte einen Ausgabewert zwischen 0 und 1 generieren. Sehr kleine Werte streben dabei gegen 0, während sehr große Werte gegen 1 streben. Die Formel sieht folgendermaßen aus:

$$f(x) = \frac{1}{1 + \exp(-b * x)} \quad (60)$$

Die Konstante b beschreibt dabei die Steilheit der Kurve. Die Ableitung der Funktion besitzt die Form $f'(x) = b * f(x)(1 - f(x))$.

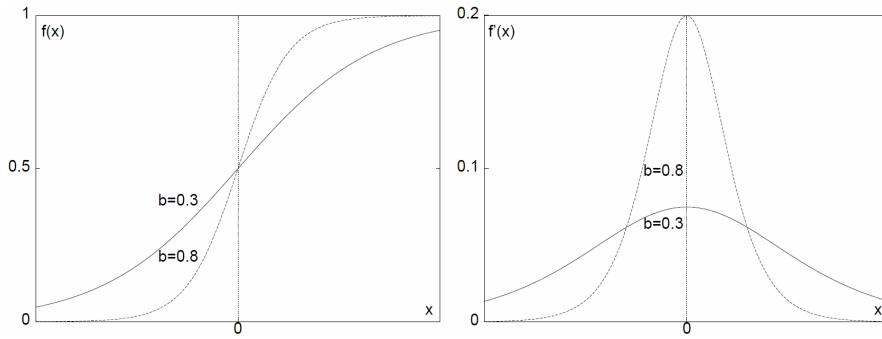


Abbildung 24: Sigmoid - Plot [10]

8. Recurrent Neural Network

Die bisher betrachteten Neuralen Netzwerke gehören zu den *forwärtsgekoppelten* Netzwerken (engl. *Feedforward Networks*). Dieser Begriff bezieht sich darauf, dass die Eingangssignale ohne Bezug auf vorherige Durchläufe ein Ergebnis generieren. Das RNN besitzt dagegen eine solche Verbindung zwischen den Neuronen derselben Schicht oder Verbindungen von Neuronen zu Neuronen einer vorgegebenen Schicht. In Abbildung 25 ist zu sehen, wie so etwas aussehen könnte:

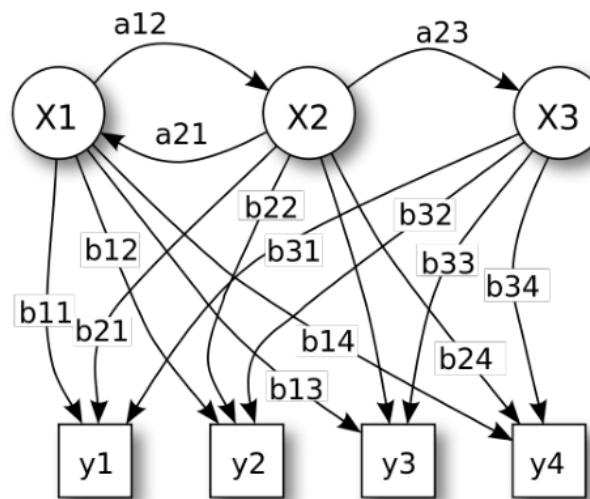


Abbildung 25: RNN - Aufbau [10]

RNNs lernen ähnlich wie die bisherigen Netzwerke mit dem wesentlichen Zusatz, dass sie Wissen aus früheren Iterationen *erinnern*, während sie den aktuellen Output generieren. In anderen Worten: Sie bekommen sowohl Eingangssignale durch den Trainingsdatensatz, als auch von sich selbst durch Rückkopplungen während der Ausführung. Deshalb ist die Reihenfolge, in der bestimmte Eingaben in das Netzwerk gespeist werden, von Bedeutung.

Diese Struktur wird in vielen unterschiedlichen Bereichen genutzt. Besonders gut eignet sie sich jedoch um bestimmte Muster zu ergänzen aber auch bei der Mustererkennung im Bereich der Sprachanalyse stellt dies ein wertvolles Werkzeug dar.

9. Fazit

9.1. Beantwortung der Leitfragen

Generell wurden folgende Leitfragen zumindest in Teilen besprochen:

- Wo können diese Verfahren eingesetzt werden, bzw. wo werden diese bereits eingesetzt?
- Welche Ansätze und Architekturen sind zur Zeit state-of-the-art?

Der Fokus dieses Abschnitts lag im wesentlichen auf dem Verständnis des *Backpropagation*-Algorithmus. Allerdings wurden ebenfalls mehrere wichtige Architekturen grob erläutert. Hierbei hat man einen kurzen Einblick in die Struktur sowie deren Verwendungszwecke in der Industrie erlangt. Für eine sehr umfangreiche Übersicht über die verschiedenen Architekturen lohnt sich ein Blick in folgende Quelle [2].

Sämtliche Materialien befinden sich im folgenden Repository:

https://github.com/derMacon/deeplearning_seminar

Literatur

- [1] 3Blue1Brown - Videokurs zur Einführung in die Neuralen Netze. https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQ0b0WTQDNU6R1_67000Dx_ZCJB-3pi. Aufgerufen am: 16-03-2020.
- [2] Übersicht - verschiedene Architekturen. <https://www.asimovinstitute.org/neural-network-zoo/>. Aufgerufen am: 22-03-2020.
- [3] Definition Klassifizierungsproblem. http://ekpwww.physik.uni-karlsruhe.de/~tkuhr/HauptseminarWS1112/Keck_handout.pdf. Aufgerufen am: 15-03-2020.
- [4] Einführung Convolutional neural network. <https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/>. Aufgerufen am: 18-03-2020.
- [5] Öffentliche Datensätze - Übersicht. <https://github.com/awesomedata/awesome-public-datasets>. Aufgerufen am: 18-03-2020.
- [6] Funktionsweise - CNN. <https://bit.ly/2QGKOEj>. Aufgerufen am: 18-03-2020.
- [7] Funktionsweise - CNN. <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1890437/>. Aufgerufen am: 18-03-2020.
- [8] Geschichte der Convolutional neuronalen Netze. <https://glassboxmedicine.com/2019/04/13/a-short-history-of-convolutional-neural-networks/>. Aufgerufen am: 18-03-2020.
- [9] Khan Academy - Partielle Ableitungen (Funktion mit zwei Eingabewerten). <https://www.youtube.com/watch?v=1CMDS4-PKKQ&t=542s>. Aufgerufen am: 16-03-2020.
- [10] Künstliche Neuronale Netzwerke und Deep Learning - Stefan Stelle. https://www.htwsaar.de/wiwi/fakultaet/personen/profile/selle-stefan/Selle2018e_Kuenstliche_Neuronale_Netzwerke.pdf/at_download/file. Aufgerufen am: 24-03-2020.
- [11] McCulloch-Pitts Neuron. <https://towardsdatascience.com/mcculloch-pitts-model-5fdf65ac5dd1>. Aufgerufen am: 14-03-2020.
- [12] Perceptron - Python Implementierung. <https://github.com/rasbt/mlxtend/blob/master/mlxtend/classifier/perceptron.py>. Aufgerufen am: 16-03-2020.
- [13] Single-Layer Neural Networks and Gradient Descent. https://sebastianraschka.com/Articles/2015_singlelayer_neurons.html. Aufgerufen am: 14-03-2020.
- [14] M.A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

A. Anhang

A.1. McCulloch-Pitts-Zelle

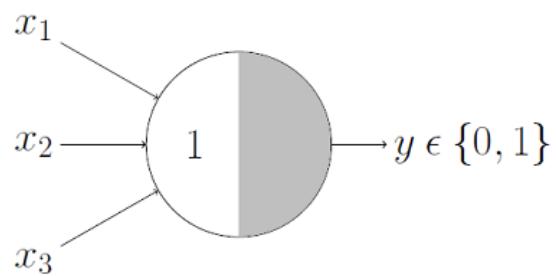


Abbildung 26: McCulloch-Pitts-Zelle - OR Gatter 2 [11]

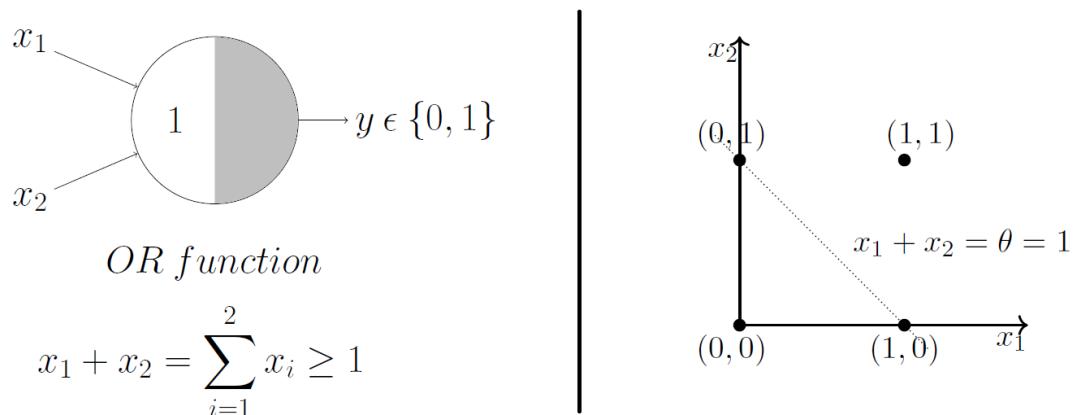


Abbildung 27: McCulloch-Pitts-Zelle - OR Gatter 3 [11]

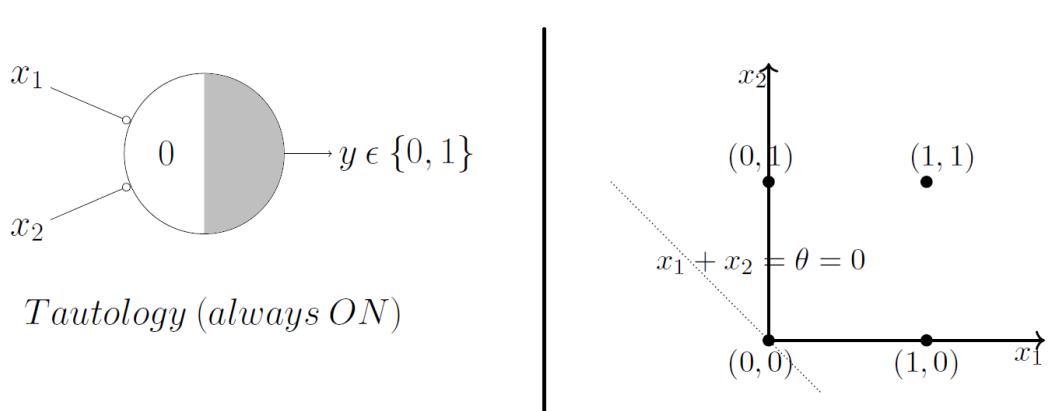


Abbildung 28: McCulloch-Pitts-Zelle - Tautologie [11]

B. Architekturen

B.1. Artenübersicht

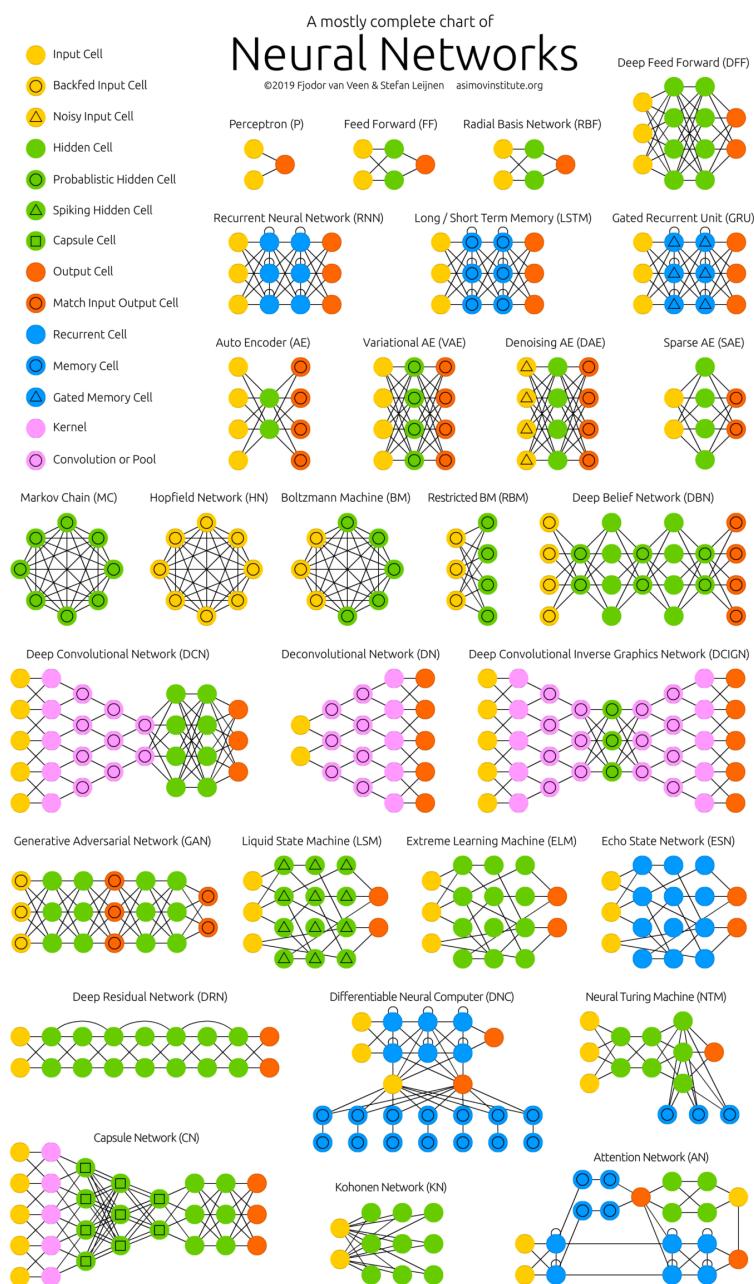


Abbildung 29: Übersicht - verschiedene Architekturen [2]