

Silas Hoffmann, inf103088

5. Fachsemester

6. Verwaltungssemester

22. März 2020

Thema 2

Seminar

im Sommersemester 2020

Dozent: Prof. Dr. Dennis Säring
Fachbereich Informatik

Fachhochschule Wedel

Inhaltsverzeichnis

I. Einführung	4
II. Geschichtliche Entwicklung	5
1. McCulloch-Pitts-Neuron	5
1.1. Funktionsweise	5
1.2. Nachteile bzw. Verbesserungspotenzial	8
2. Perceptron	9
2.1. Aufbau und Notation	9
2.2. Lernregel	11
3. Adeline	14
3.1. Aufbau	14
3.2. Lernalgorithmus	14
4. Convolutional Neural Network	23
4.1. Geschichte	23
4.2. Funktionsweise	25
5. Fazit	29
III. Aktuelle Entwicklung	30
6. Backpropagation	30
6.1. Notation	30
6.2. Fundamentale Gleichungen	32
6.3. Fehler (Gleichung 30)	32
A. Anhang	ii
A.1. McCulloch-Pitts-Zelle	ii
B. Architekturen	iv

Abbildungsverzeichnis

1.	McCulloch-Pitts-Zellne - Genereller Aufbau	6
2.	McCulloch-Pitts-Zelle: Aufbau und Klassifizierung	7
3.	Biologische Neuronen - Notation	7
4.	McCulloch-Pitts-Zelle - AND Gatter	7
5.	McCulloch-Pitts-Zelle - OR Gatter	8
6.	Perceptron - drei Eingabewerte	9
7.	Perceptron - Einheits-Sprungfunktion	10
8.	Perceptron - Modelansicht	11
9.	Perceptor - Problematische Klassifizierung	13
10.	Adeline - Aufbau	14
11.	Gradientenverfahren mit eindimensionaler Kostenfunktion	15
12.	Gradientenverfahren - unterschiedliche Lernraten	16
13.	Gradientenverfahren - zweidimensionaler Eingabevektor	17
14.	Partielle Ableitung 1	18
15.	Partielle Ableitung 2	19
16.	Gradientenverfahren - Vergleich	22
17.	Simple Cell - Beispiel	23
18.	Vergleich - Simple und Complex Cell	24
19.	Vergleich - Darstellung Mensch und Maschine	25
20.	Überblick - CNN Verarbeitungsschritte	26
21.	CNN - Convolution Layer	26
22.	CNN - Filter	27
23.	Notation	30
24.	Notation	31
25.	McCulloch-Pitts-Zelle - OR Gatter 2	ii
26.	McCulloch-Pitts-Zelle - OR Gatter 3	ii
27.	McCulloch-Pitts-Zelle - Tautologie	iii
28.	Übersicht - verschiedene Architekturen	iv

Teil I.

Einführung

Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

Teil II.

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 vorher nie in Betracht gezogen hatte. Dennoch hat es mich doch überrascht wie „alt“ diese Technologie zumindest in ihren Grundzügen in manchen Teilen doch bereits ist. Ihre Anfänge fand sie schon Mitte der 50er Jahre als zum Beispiel diverse Experten auf diesem Gebiet das erste Mal zusammenkamen um im Zuge der *Dartmouth Konferenz* über dieses Thema zu diskutieren. Aber auch bekannte Persönlichkeiten des Feldes, wie zum Beispiel *Von Neumann*¹ erkannten früh die möglichen Zusammenhänge.

Im folgenden Abschnitt werde ich etwas auf die geschichtlichen Aspekte von neuronalen Netzen eingehen. Hierbei werden insbesondere die generellen Aspekte der generellen 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?

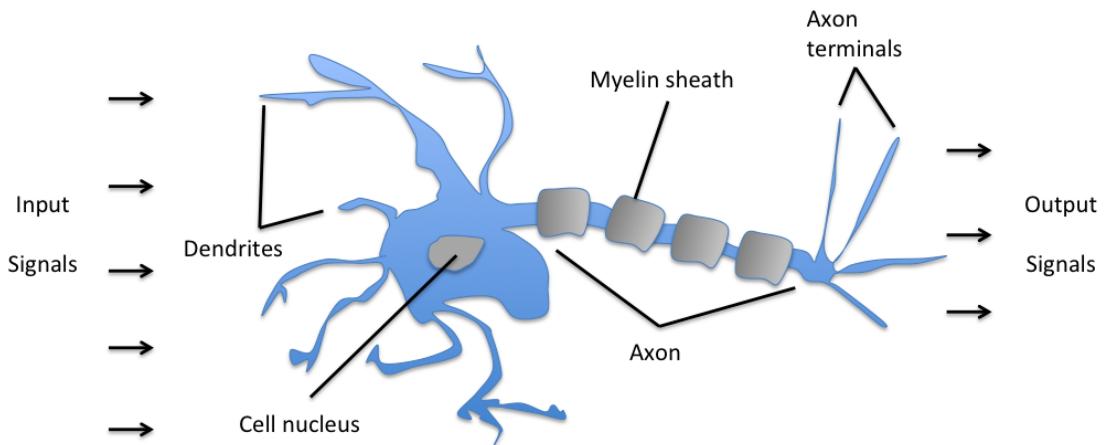
1. McCulloch-Pitts-Neuron

1.1. Funktionsweise

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

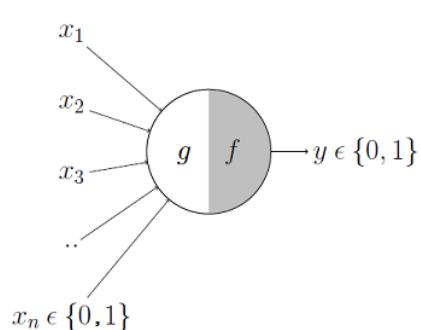
Abbildung 1: McCulloch-Pitts-Zellne - Genereller Aufbau [8]

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 ist 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.* [2] oder nicht. Der grobe Aufbau eines sogenannten *McCulloch-Pitts-Zelle* ist in ?? zu sehen. Erwähnt sei auch noch, dass man mit diesem Modell lediglich binäre Klassifizierungen mittels einer linearen Entscheidungsfunktion / Aktivierungsfunktion durchführen kann (siehe 2)

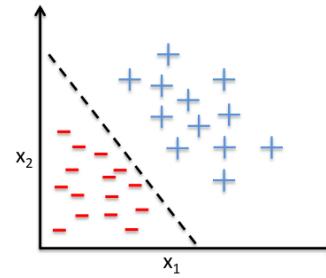
Das Modell kann beliebig viele Input-werte aufweisen. Wichtig hierbei: Sie dürfen nur boolescher Natur sein (nur falsch oder wahr). Bei gegebenen Werten führt das Neuron selbst zwei Arbeitsschritte durch:

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 wird 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 [8]

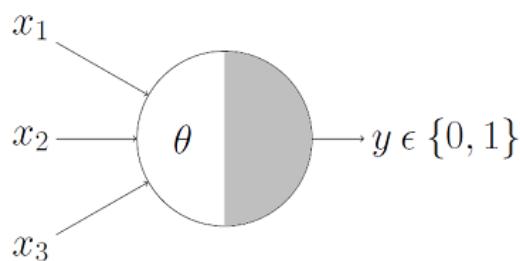


Abbildung 3: Biologische Neuronen - Notation [8]

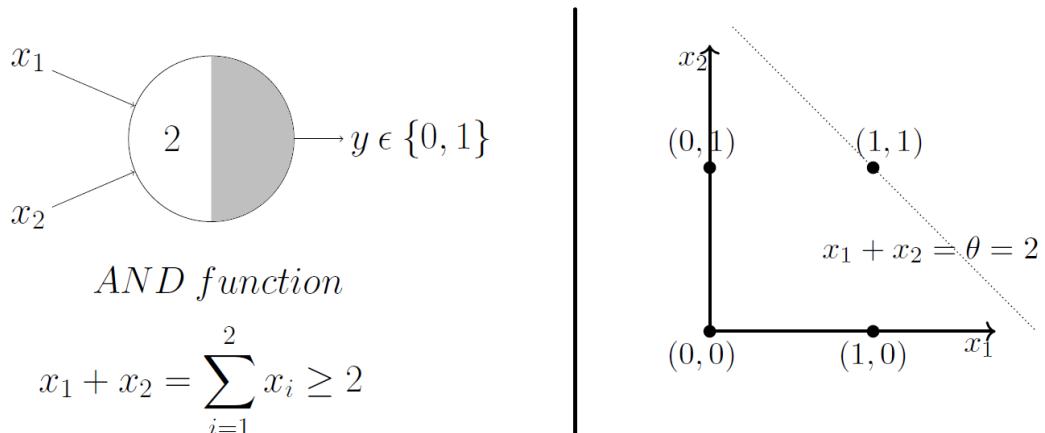
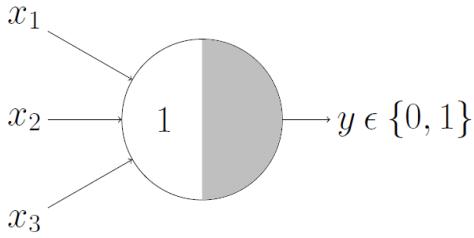


Abbildung 4: McCulloch-Pitts-Zelle - AND Gatter [8]



OR function

$$x_1 + x_2 + x_3 = \sum_{i=1}^3 x_i \geq 1$$

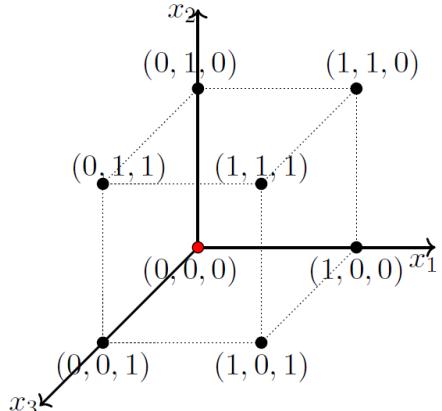


Abbildung 5: McCulloch-Pitts-Zelle - OR Gatter [8]

1.2. Nachteile bzw. Verbesserungspotenzial

- Dieses Modell erlaubt wie gesagt nur boolesche Eingabewerte, viele Modelle fordern 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 Priorisierungsmöglichkeit zwischen den einzelnen Eingabewerten. Jeder hat einen gleichgroßen Einfluss auf das Ergebnis, so etwas wie ein Auschlußkriterium gibt es hierbei also nicht.
- Es ist nicht möglich *gedeckelt* Gatter wie zum Beispiel ein *XOR* abzubilden. Bei einem Neuron mit zwei Input 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.

2. Perceptron

2.1. Aufbau und Notation

Im Jahr 1958 entwickelte der US-amerikanischer Psychologe und Informatiker Frank Rosenblatt das sogenannte *Perceptron*. Dieses stellt das älteste neurale 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.

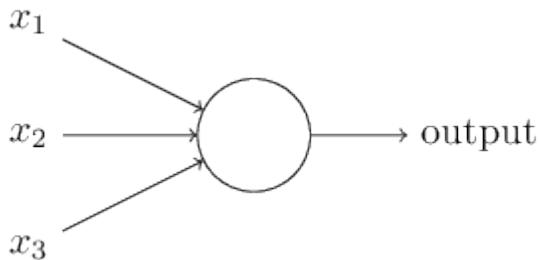


Abbildung 6: Perceptron - drei Eingabewerte [11]

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).

$$g(\mathbf{z}) = \begin{cases} 0 & \text{if } \mathbf{z} \leq \theta \\ 1 & \text{if } \mathbf{z} > \theta \end{cases} \quad (1)$$

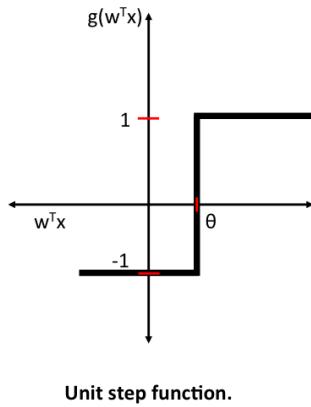
wobei gilt

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

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

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

Geplottet sieht die Aktivierungsfunktion g übrigens folgendermaßen aus (siehe ??):



Unit step function.

Abbildung 7: Perceptron - Einheits-Sprungfunktion [8]

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

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

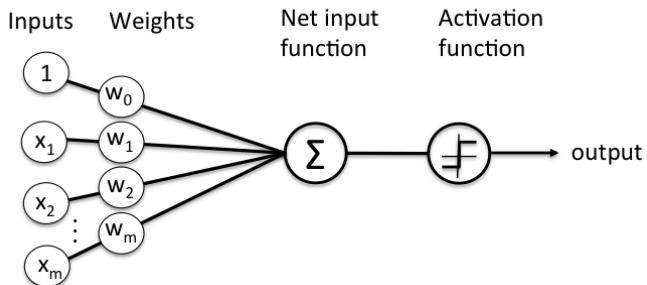
wobei gilt

$$\mathbf{z} = \mathbf{w}_0 \mathbf{x}_0 + w_1 x_1 + \cdots + w_m x_m = \sum_{j=1}^m x_j w_j = \mathbf{w}^T \mathbf{x} \quad (5)$$

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:

Letztendlich sei jedoch erwähnt, dass es in der Literatur auch öfters eine Darstellung mittels eines *Bias* gibt, siehe Gleichung 6 [11]

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



Schematic of Rosenblatt's perceptron.

Abbildung 8: Perceptron - Modelansicht [8]

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 Null.
- Für jeden Datensatz der Menge:
 - Berechne den Ausgabe-Wert des Systems
 - Gleiche die Gewichte an

Der genannte Ausgabewert wird wie bereits angedeutet über die Aktivierungsfunktion des Modells bestimmt. Die einzelnen Komponenten des *Gewichtevektors* 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 Dreieck 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 eigentliche Formel 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 betrachten 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 8 auf Seite 11 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 Trainingsdatensatz² würde die Lernregel folgendermaßen aussehen. Hervorzuheben sei jedoch, dass alle Gewichte gleichzeitig angeglichen werden.

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

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 weiteren 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 [9] zu finden. Ich werde hier jedoch nicht weiter auf die Details der Implementierung eingehen.

Wie schon die McCulloch-Pitts Zelle (siehe Abbildung 2) ist das Perceptron nur in der Lage mittels einer linearen Klassifizierungs-Funktion die zwei unterschiedlichen Gruppen auseinander zu halten (siehe Abbildung 9).

Hier noch einmal extra aufgeführt, die letztendliche Formel zum Berechnen einer einzelnen Gradientenkomponente.

²Ein zweidimensionaler Datenpunkt besitzt einen zweidimensionalen Eingabevektor (also genau zwei Eingabewerte).

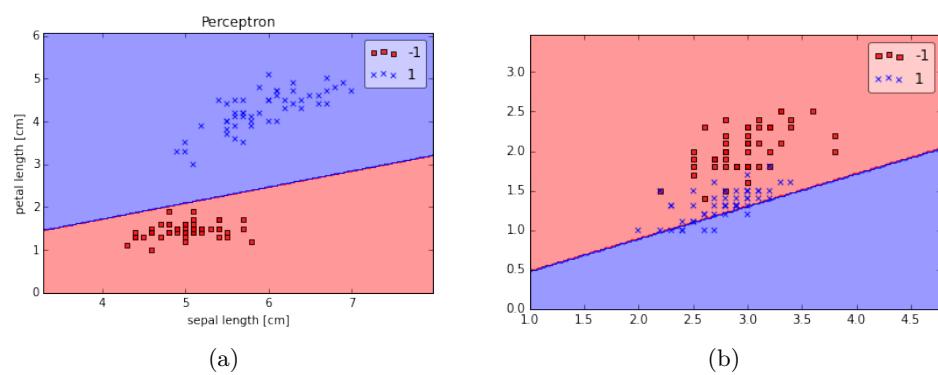


Abbildung 9: Perceptor - Problematische Klassifizierung [8]

3. Adeline

3.1. Aufbau

Im 1959 entwickelten der Stanford Professor Bernard Widrow und der Elektroingenieur Marcian Edward Hoff das sogenannte *Adeline-Modell*. Der Name ist ein Akronym für *AD-Adaptive LINear Element*. Dieses Modell basiert auf dem Perceptron mit dem Unterschied, dass dieses Modell auf die Einheits-Sprungfunktion, wie sie das Perceptron verwendet, bei der Angleichung der Gewichte verzichtet. Es wird stattdessen eine lineare Aktivierungsfunktion $g(\mathbf{z})$ verwendet welche in diesem Fall erstmal mit der Identitätsfunktion besetzt wird (es gilt also $g(\mathbf{w}^T \mathbf{x}) = \mathbf{w}^T \mathbf{x}$). 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.

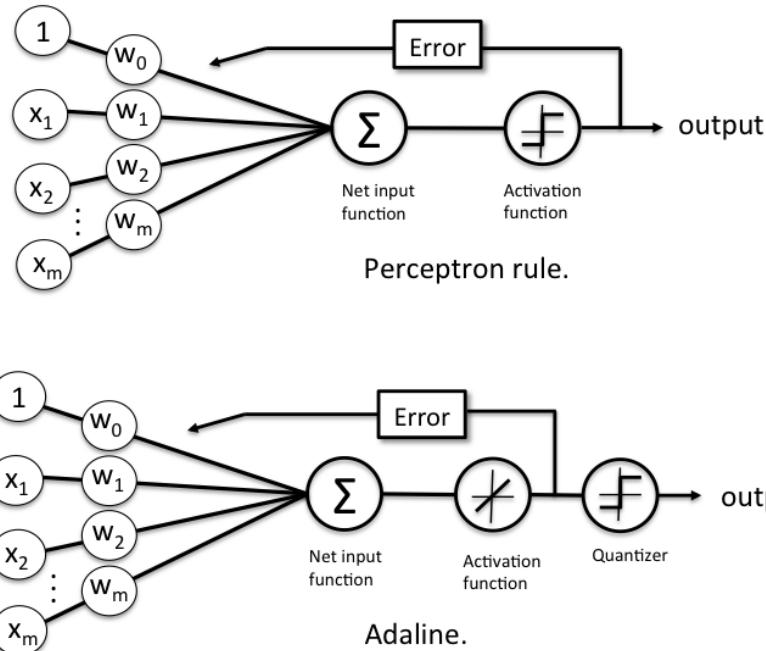


Abbildung 10: Adaline - Aufbau [8]

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.

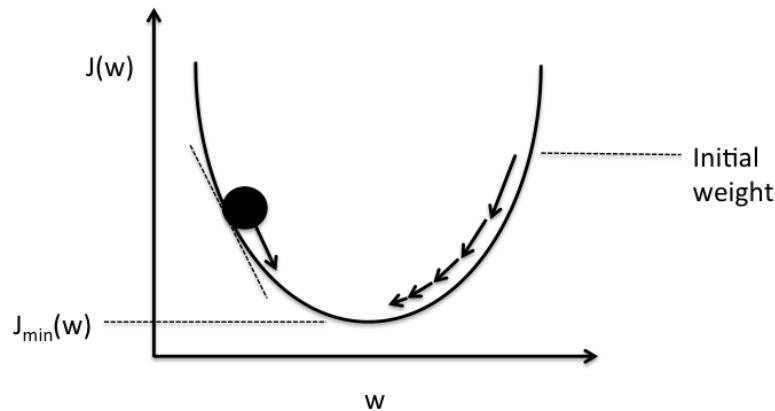
Gradientenverfahren Der wesentliche Nachteil der Einheits-Sprungfunktion ist der, dass sie nicht stetig und damit auch nicht differenzierbar ist. Deswegen hat man sich beim Lernalgorithmus des Adeline-Modells dazu entschieden stattdessen die Identitätsfunktion zu verwenden.

Es wird zuerst eine Kostenfunktion $J(\mathbf{w})$ definiert die minimiert werden soll. Die Kostenfunktion wird durch die *Regressionsquadratsumme*³ definiert. Die Formel sieht folgendermaßen aus:

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

Wichtig hierbei, der Vorfaktor $\frac{1}{2}$ gehört nicht zur herkömmlichen Regressionsquadratsumme, wurde hier jedoch hinzugefügt um später einfach 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 11.



Schematic of gradient descent.

Abbildung 11: Gradientenverfahren mit eindimensionaler Kostenfunktion [8]

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

³engl. sum of squared errors (SSE)

verstehen. Wie weit in einem Schritt gearbeitet wird ist von fundamentaler Bedeutung. In Abbildung 12 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.

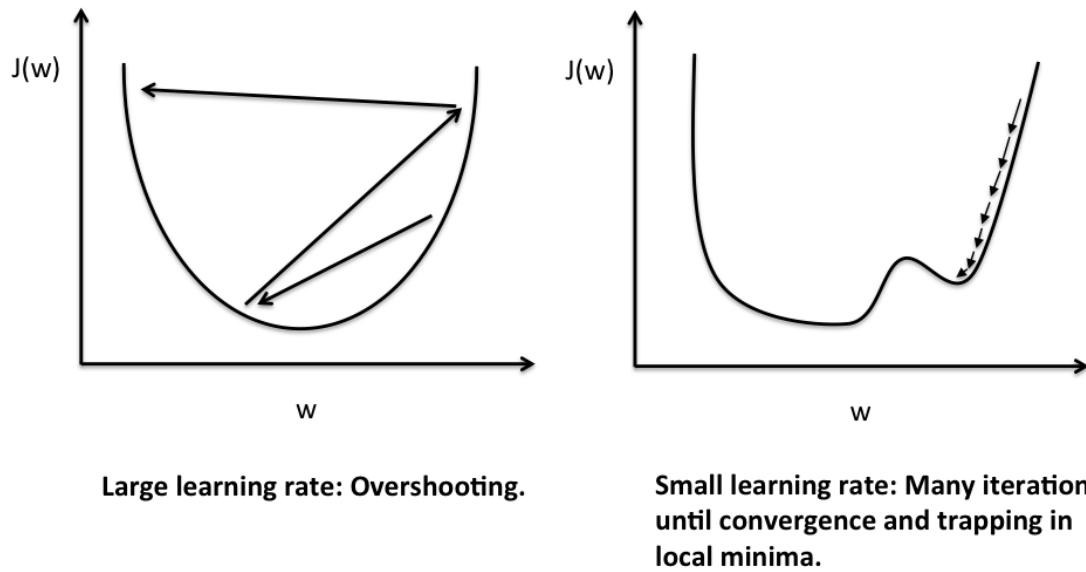


Abbildung 12: Gradientenverfahren - unterschiedliche Lernraten [8]

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 wohl auf eine Kostenfunktion mit einem mehrdimensionalen Vektor anpassen muss. Mit einem zweidimensionalen Eingabevektor ist dies noch relativ gut darstellbar (siehe Abbildung 13).

Auch bei Funktionen mit mehreren Eingabewerten ist es möglich an einem betrachteten Eingabevektor die Steigung zu bestimmen, diese nennt sich hier jedoch *Gradientenvektor*. Bei der gerade betrachteten Abbildung ist 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 bei den anderen betrachteten Modellen wieder die Notation:

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

Δw stellt den angedeutete Gradientenvektor dar. Im mehrdimensionalen Raum ist diese Gewichtsveränderung allgemein Fall für den kompletten Gewichtsvektor und für

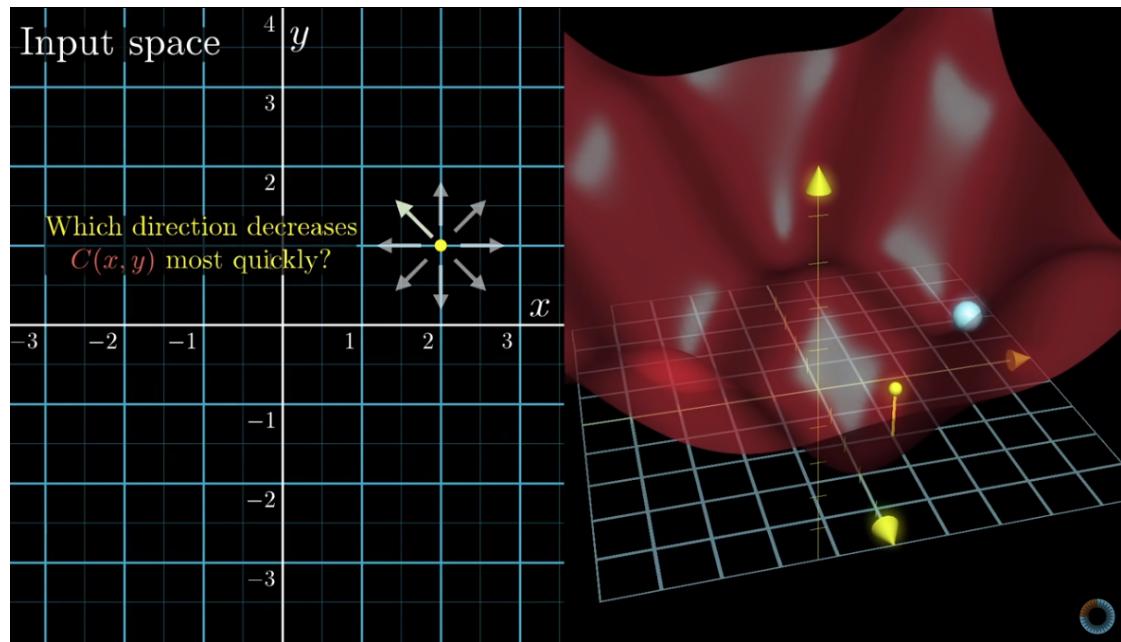


Abbildung 13: Gradientenverfahren - zweidimensionaler Eingabevektor [8]

den speziellen Fall mit einem einzelnen Gewicht folgendermaßen definiert.

Regel 3.1: Gradient

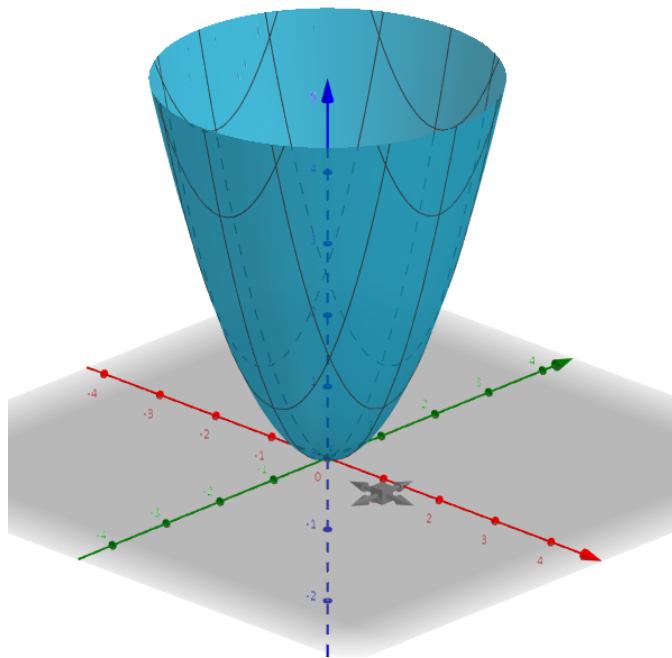
Allgemein:

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

Für die jeweiligen Gewichte:

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

Exkurs - Partielle Ableitungen [7] Da es sich im Beispiel um einen mehrdimensionalen Eingabevektor handelt muss man zum hierbei 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 14).

Abbildung 14: Partielle Ableitung 1⁴

Wann man nun einen Punkt auf der Oberfläche der Funktion nimmt 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 15. Hier wurde eine „rote Ebene“ eingeführt. Dies kann als *Blickwinkel* betrachtet werden. Wenn man nun die Schnittgerade betrachtet welche durch die Ebene und die Funktion aufgestellt wurde ist es möglich auf dieser wiederum einen Punkt mit seiner zugehörigen Steigung zu ermitteln da man den *Blickwinkel* kennt. 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 Aufteilens 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 Nenner 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.

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

⁴Mit Geogebra 3d erstellt: <https://www.geogebra.org/3d?lang=de>

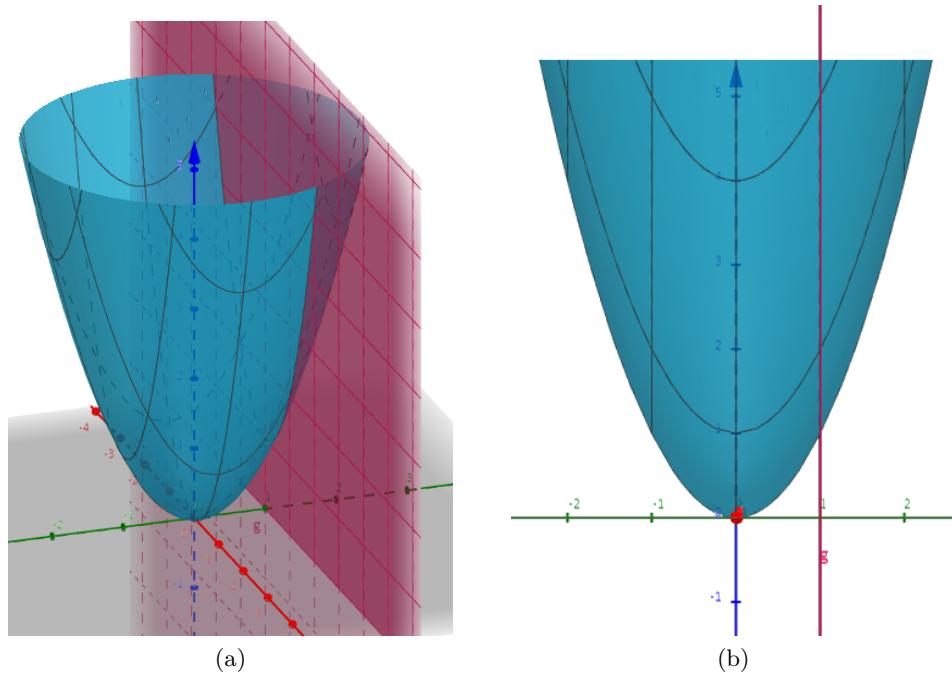


Abbildung 15: Partielle Ableitung 2

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

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 Gewichts ableiten welches wir im Algorithmus gerade behandeln. 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 Faktor konstante Faktor zwei 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 die Vektormultiplikation $\mathbf{w}^T \mathbf{x}$. 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 den Faktor $-x_j^{(i)}$ 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 3.2) 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 aller erste Formel (siehe 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 [9].

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 Trainingsdatensätzen 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 16 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.

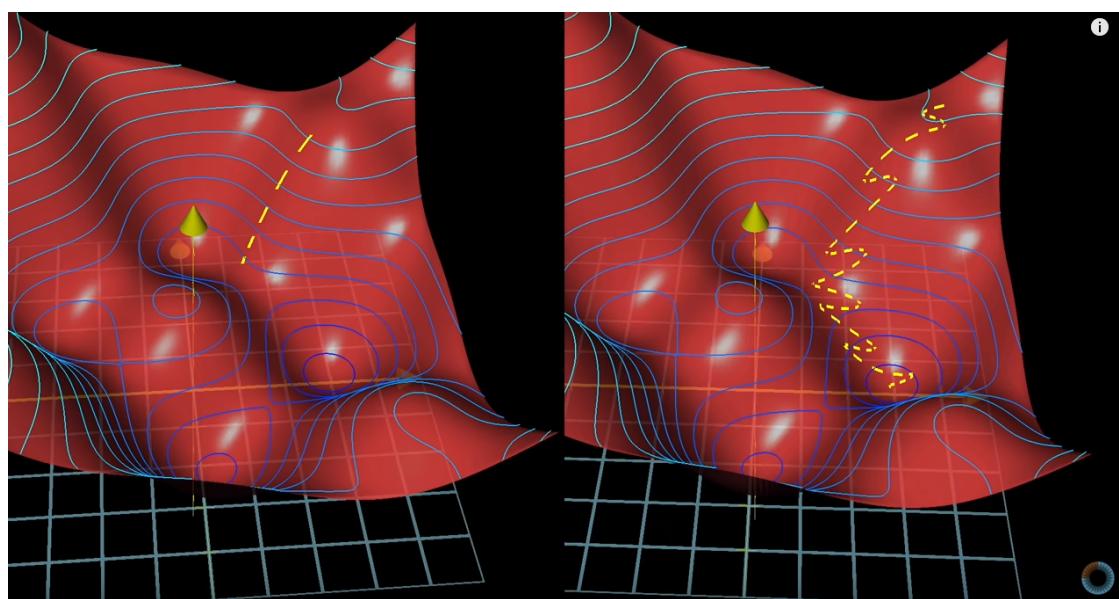


Abbildung 16: Gradientenverfahren - Vergleich [1]

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 erkennen kann.

4.1. Geschichte

Zellarten Im Jahr 19459 beschrieben die beiden Neurophysiologe 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 Cortex verwendet werden.

- Die *simple cells* können einfache Kanten und Balken mit einer bestimmten Orientierung erkennen. Abbildung 17 zeigt welche Art von Formen von diesen Zellen erkannt werden können. Ein solche Zelle könnte zum Beispiel in der Lage sein einen Balken am unteren Bildrand als solchen zu erkennen.
- Eine *komplexe* ist ebenfalls dazu in der Lage diese Formen zu erkennen allerdings mit dem Zusatz, dass sie in der Lage ist 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 Positionsunabhängigkeit eines Musters wird *spatial invariance* genannt (zu deutsch „räumliche Invarianz“).

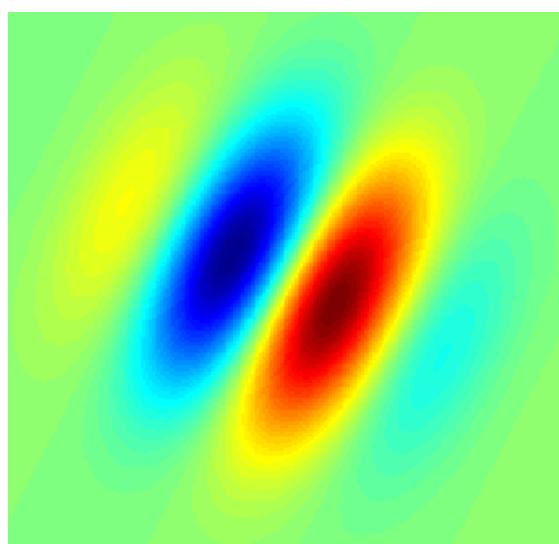


Abbildung 17: Simple Cell - Beispiel [6]

Einige Jahre später (1962), beschrieben die beiden Wissenschaftler wie solche komplexen Zellen die Eigenschaft der *partial invariance* erreichen. Diese Zellen summieren

die Ausgaben von mehreren *simple cells* auf. Diese Zellen sind auf die gleichen Formen spezialisiert, analysieren jedoch jeweils einen unterschiedlichen Teil des Bildes (englisch *receptive fields*, siehe Abbildung 18). 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.

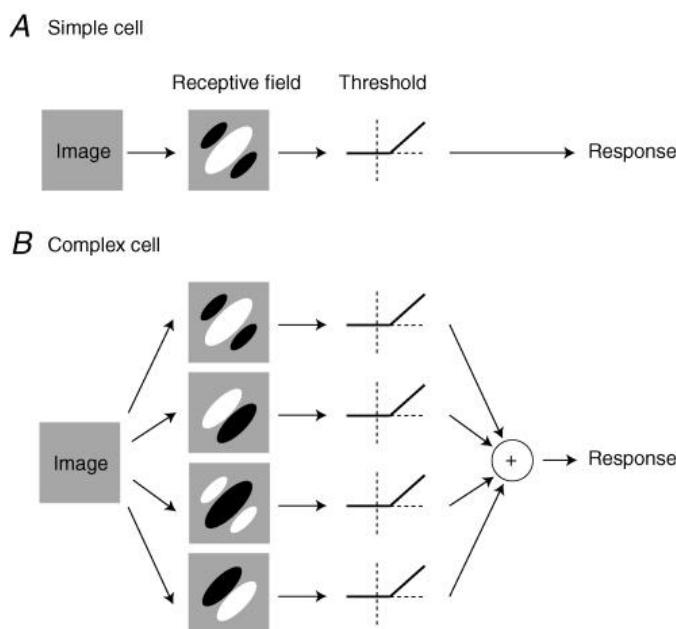


Abbildung 18: Vergleich - Simple und Complex Cell [?]

In den 1980er 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* einführt. 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 Ausarbeitung. 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 der die *MNIST database of handwritten digits*. Diese Datenbank enthält Bilder von handgeschriebenen Ziffern welche jeweils mit einem entsprechenden Label versehen wurden welche vom Netz verwendet werden um die Kostenfunktion auf-

zustellen⁵. 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 erklärt.

Todo Referenz auf Erklärung einbinden.

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 auch durchaus mehr als eine Klasse pro Bild sein. Das zu kategorisierende Bild wird in Form einer Matrix verarbeitet. In der folgenden Darstellung kann man grob sehen wie sowas in etwa aussehen könnte (siehe Abbildung ??).



```

08 02 22 97 38 15 00 75 04 05 07 78 52 12 50 77 91 08
49 49 99 40 17 81 18 57 60 87 17 40 98 43 69 45 04 56 62 00
81 49 31 73 55 79 14 29 93 71 40 67 53 88 34 03 49 13 36 45
52 70 95 23 04 60 11 42 69 24 68 54 01 32 56 71 37 02 36 91
23 31 14 71 51 47 63 89 41 92 36 54 22 40 44 28 66 33 13 80
24 47 32 60 99 03 61 02 44 75 33 53 78 36 84 20 35 17 12 50
32 98 81 21 64 23 67 10 26 38 40 67 59 54 70 66 18 38 64 70
67 26 20 61 02 62 12 20 95 63 94 39 63 08 40 91 66 49 94 21
24 55 58 05 66 73 99 26 97 27 78 79 96 82 14 88 34 89 63 72
21 36 23 09 75 00 76 44 20 45 35 14 00 61 83 97 34 91 33 95
78 17 53 28 22 75 31 67 15 94 03 80 04 62 16 14 09 83 56 92
16 39 05 42 96 35 31 47 55 58 88 24 00 17 54 24 36 29 85 57
86 56 00 45 71 89 07 05 44 44 37 44 60 21 58 51 54 17 58
19 60 81 60 05 94 47 69 28 75 92 13 86 52 17 77 04 69 55 40
04 52 05 83 97 35 99 16 07 97 57 32 16 24 26 79 33 27 98 46
88 36 68 87 57 62 20 72 03 46 33 67 46 55 12 32 63 93 53 69
04 92 16 77 38 25 39 11 24 94 72 18 08 46 29 32 49 62 76 36
20 69 36 41 72 30 23 88 34 62 99 69 82 67 59 85 74 04 36 16
20 73 35 29 78 31 90 01 74 31 99 71 48 84 81 16 23 57 05 54
01 70 54 71 83 51 54 69 16 92 33 48 61 43 52 01 89 19 67 48

```

[What We See](#)

[What Computers See](#)

Abbildung 19: Vergleich - Darstellung Mensch und Maschine [3]

Die rechte Darstellung ist hierbei auch noch unvollständig, denn es handelt sich bei dem Bild um ein Farbbild. In der Praxis ist 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 auf *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*) und Aggregations-Schichten (*Pooling Layer*). Diese wiederholen sich abwechselnd. In Abbildung 20 ist zu erkennen wie die beschriebenen Zwischenschritte der Verarbeitung grob aussehen können.

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 x 5. Diese spannt ein kleines „Fenster“ über der Matrix auf. Dieses Fenster wandert beziehungsweise scannt anschließend mit einer definierten Schrittwei-

⁵Es sei ebenfalls noch kurz erwähnt, dass es zu sehr vielen Themengebieten derartige Datenbanken gibt. Eine umfangreiche Liste von frei verfügbaren Quellen ist hier [4] zu finden.

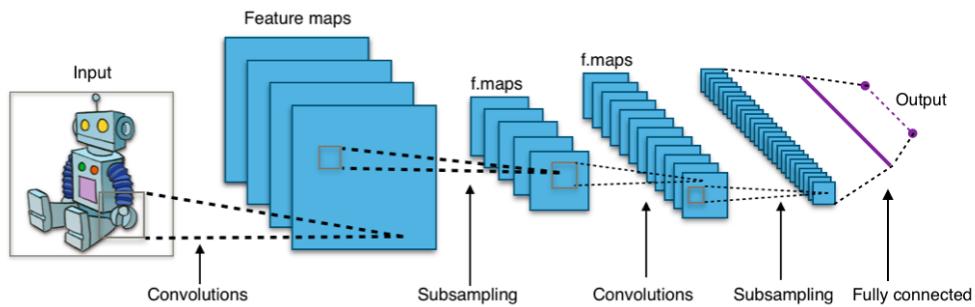


Abbildung 20: Überblick - CNN Verarbeitungsschritte [?]

te zeilenweise über 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 ein Neuron in der nächsten Schicht. Die Abbildung 21 zeigt in relativ verständlicher Weise 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 [?]*.

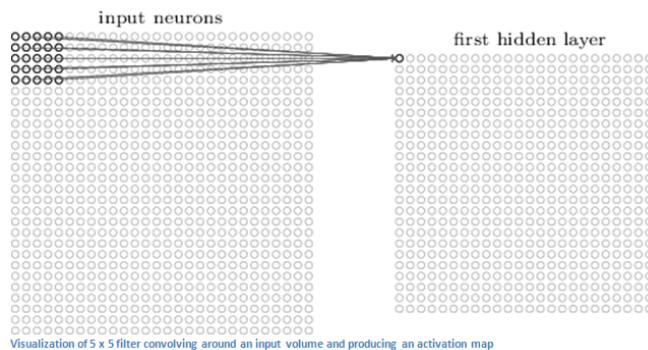


Abbildung 21: CNN - Convolution Layer [?]

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 Bit oder 16 Filtern verwendet. 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 22). 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.

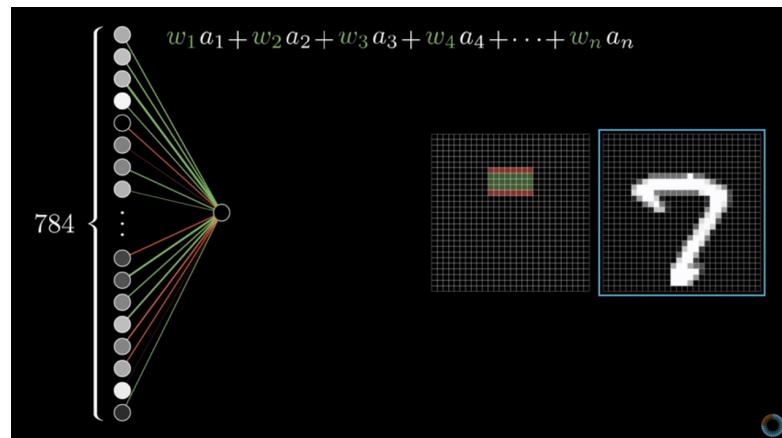


Abbildung 22: CNN - Filter [1]

Um diese Ziffer 7 als eine solche zu erkennen muss das Netz in der Lage 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 Null darstellen. Wenn man nun die erhaltenen Pixelwerte (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 sein sollte. Durch diesen Sachverhalt ist es möglich eine diverse bestimmte einfache Eigenschaften des Bildes zu erkennen.

Pro zu erkennenden Merkmal gibt es in einem CNN einen Filter geben. Jedes Betrachtungsfenster dieses Filters würde dann die erwarteten Farbwerte für die jeweilige Ziffer wiederspiegeln.

Die Erkenntnisse dieser *low-Level* Filter können dann wiederum mit 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* [?]. Generell gilt, während die Grö-

ße des Inputs durch die Faltungen und das Pooling immer weiter reduziert wird, wird die Anzahl der Filter zur Erkennung von übergeordneten Signalen zunehmend erhöht.

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 werden:

- fractuional 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 bereits beschriebenen Schichten erkannt wurden. In dieser Schicht werden sämtliche Neuronen mit allen Ein- und 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 es 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 Das genutzte Verfahren der *Backpropagation* ist von fundamentaler Bedeutung für alle aktuellen Architekturen, daher bekommt er eine eigenen Section (siehe todo Referenz einfügen).

5. Fazit

Generell wurden folgende Leitfragen zumindest in Teilen besprochen:

- 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?

Der Begriff des Deeplearnings wurde in diesem Abschnitt noch überhaupt nicht besprochen da es sich hierbei um eine sehr weiterentwickelte Form des Begriff *Machine Learning* handelt. Dies wird allerdings im kommenden Abschnitt besprochen (todo Referenz). Generell wurden jedoch die wesentlichen Vorstufen grob angerissen. Wir haben die Ursprünge der ersten künstlichen Neuronen (siehe MPN 1, Perceptron 2) gesehen und wie diese auf der Biologie des Menschen basieren. Darüber hinaus wurden die Verbesserungen der ersten Versuche vorgestellt (siehe Adeline 3). Außerdem haben ich die ersten Anfänge des künstlichen Lernens mithilfe von Lernalgorithmen sowie die Mathematik dahinter beschrieben (siehe 3.2). Um auch einen kleinen praktischen Einschnitt zu haben wurde ebenfalls grob beschrieben wie ein Rechner in der Lage ist bestimmte Merkmale eines Bildes zu erkennen (siehe CNN 4).

Teil III.

Aktuelle Entwicklung

- 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

Um in einem mehrschichtigen Netz effizient die Kostenfunktion zu minimieren wird in vielen Fällen der *Backpropagation*-Algorithmus verwendet. Dieser 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 kann.

Gemeinhin wird die Notation wie sie in Abbildung 23 zu sehen ist, für ein Gewicht verwendet. In Abbildung 24 steht das b_j^l für den Schwellwert (*Bias*) am Neuron mit dem Index j im Layer mit dem Index l . Selbes 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).

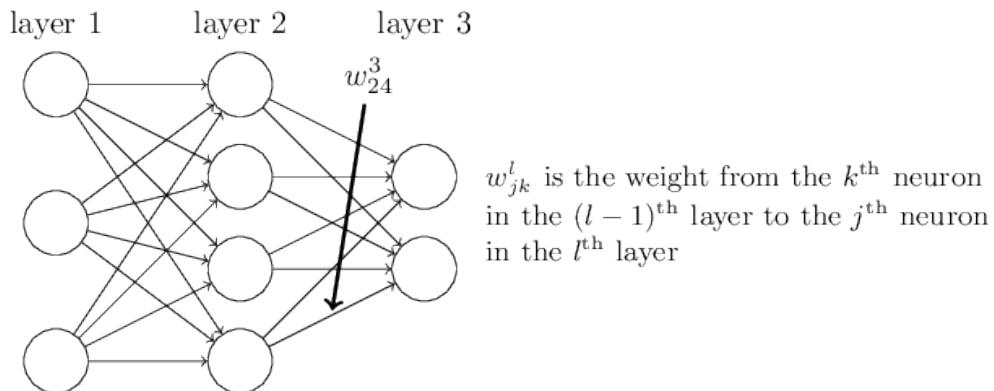


Abbildung 23: Notation [11]

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

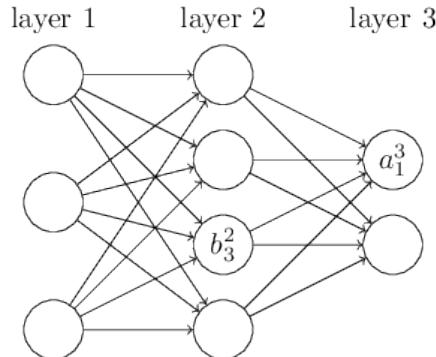


Abbildung 24: Notation [11]

Diese Formel sollte bereits aus den vorherigen Kapiteln bekannt sein. Es wird hierbei lediglich eine Vektormultiplikation der beiden eingehenden Gewichtsvektors und Addition mit den Aktivierungsvektoren durchgeführt. Die generierte Ausgabe wird mit dem Schwellwert verrechnet und in eine Aktivierungsfunktion (wie zum Beispiel der Sigmoid-Funktion) gesteckt.

Um später einfacher mit all diesen Werten zu rechnen wird versucht die gegebenen Werte in eine Matrix- beziehungsweise Vektor-Darstellungsform zu bringen. Da Neuron mehrere ausgehende „Pfade“ besitzen wird hierbei eine Matrix geformt. Das bisher beschriebene Gewicht w_{jk}^l befindet sich hierbei in der Zeile mit dem Index j und Spaltenindex k . Da sich sowohl der Schwellwert als auch die Aktivierung lediglich 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 vektorisieren⁶. 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 umgeschrieben 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

⁶Klarer Verweis auf Nielsen Buch, Kapitel über Backpropagation im Detail [11]

zu schaffen wird oftmals die Aktivierung einer Schicht l aus der Funktion herausgezogen und mit dem Buchstaben z versehen. z^l kann nun in die Aktivierungsfunktion σ eingesetzt werden. Im folgenden habe ich noch einmal alle bisherigen Erkenntnisse zusammengefasst:

Regel 6.1: Backpropagation - Notation

Bisherige Schreibweise:

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

Zusammengefasste Form:

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

Gewichtete Eingabe:

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

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

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

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

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

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

6.3. 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 σz_j^l wird ein *Fehler* Δz_j^l hinzugefügt. Allgemein wird der Fehler eines einzelnen Neurons dadurch folgendermaßen beschrieben:

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

Wie wir bereits beim Gradientenabstieg gesehen haben, beschreibt diese partielle Ableitung die „Steigung“ der Kostenfunktion im Bezug auf den die Ausgabe z_j^l . Dieser Wert dient im späteren Verlauf allerdings lediglich als Zwischenschritt, da wir im Endeffekt an den Gradienten bezüglich der Gewichte sowie den Schwellwerten interessiert sind. Im Nachhinein wird daher versucht den Fehler $\partial C / \partial z_j^l$ auf die Gewichte $\partial C / \partial w_j^l$ und Bias $\partial C / \partial b_j^l$ zurückzuführen.

Die komponentenweise Darstellung für die Berechnung des Fehlers auf der Ausgabeschicht δ^L sieht folgendermaßen aus:

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

Der vordere Teil $\partial C / \partial a_j^L$ beschreibt wie sich die Kostenfunktion bezüglich eines Neurons mit dem Index j auf der Ausgabeschicht verhält (Steigung der Kostenfunktion nach a_j^L). Wenn das Neuron nicht sehr viel *Einfluss* auf die Kostenfunktion nimmt, bleibt dieser Faktor klein und das Ergebnisfehler fällt minimal aus. Der hintere Teil beschreibt die Ableitung der Aktivierungsfunktion und die Steigung an der gegebenen Stelle z_j^L . Da man hierbei auch wieder versuchen möchte die Formeln möglichst allgemein zu halten wird der Fehler auf wieder als Vektor der Ausgabeschicht definiert. Das sieht dann so aus:

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \quad (37)$$

Dies ist auch die Darstellung wie sie Anfang des Abschnitts verwendet wurde (siehe 6.2). $\nabla_a C$ stellt dabei den Vektor dar dessen partielle Ableitung $\partial C / \partial a_j^L$ entsprechen. Es ist also lediglich eine andere Schreibweise für die partielle Ableitung bezogen auf die komplette Ausgabeschicht. Wenn wir für die Kostenfunktion wie schon in früheren Kapiteln die quadratische Kostenfunktion wählen entspricht C gleich $C = \frac{1}{2} \sum_j (y_j - a_j^L)^2$. Nach a_j^L abgeleitet entspricht dies dann wiederum $\partial C / \partial a_j^L = (a_j^L - y_j)$. Die Herleitung ist sehr ähnlich zu der des Gradienten (siehe 3.2), hier fehlt lediglich die Iteration über die kompletten Trainingsdatensätze.

Der entstandene Vektor ($\nabla_a C$) wird dann komponentenweise mit der Ableitung der Aktivierungsfunktion multipliziert. Da diese vektorisiert wurde entspricht das Ergebnis von $\sigma'(z^L)$ ebenfalls wieder einem Vektor was diese Rechenoperation ermöglicht. Eine andere Schreibweise ist daher folgende Gleichung:

$$\delta^L = (a^L - y) \odot \sigma'(z^L) \quad (38)$$

Hierbei wurde lediglich die Ableitung eingesetzt.

Literatur

- [1] 3Blue1Brown - Videokurs zur Einführung in die Neuralen Netze. https://www.youtube.com/watch?v=aircArUvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi. Aufgerufen am: 16-03-2020.
- [2] Definition Klassifizierungssproblem. http://ekpwww.physik.uni-karlsruhe.de/~tkuhr/HauptseminarWS1112/Keck_handout.pdf. Aufgerufen am: 15-03-2020.
- [3] Einführung Convolutional neural network). <https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/>. Aufgerufen am: 18-03-2020.
- [4] Öffentliche Datensätze - Übersicht). <https://github.com/awesomedata/awesome-public-datasets>. Aufgerufen am: 18-03-2020.
- [5] Funktionsweise von Convolutional neuronalen Netzen). <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1890437/>. Aufgerufen am: 18-03-2020.
- [6] Geschichte der Convolutional neuronalen Netze). <https://glassboxmedicine.com/2019/04/13/a-short-history-of-convolutional-neural-networks/>. Aufgerufen am: 18-03-2020.
- [7] Khan Academy - Partielle Ableitungen (Funktion mit zwei Eingabewerten). <https://www.youtube.com/watch?v=1CMDS4-PKKQ&t=542s>. Aufgerufen am: 16-03-2020.
- [8] McCulloch-Pitts Neuron. <https://towardsdatascience.com/mcculloch-pitts-model-5fdf65ac5dd1>. Aufgerufen am: 14-03-2020.
- [9] Perceptor - Python Implementierung. <https://github.com/rasbt/mlxtend/blob/master/mlxtend/classifier/perceptron.py>. Aufgerufen am: 16-03-2020.
- [10] Single-Layer Neural Networks and Gradient Descent. https://sebastianraschka.com/Articles/2015_singlelayer_neurons.html. Aufgerufen am: 14-03-2020.
- [11] M.A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

A. Anhang

A.1. McCulloch-Pitts-Zelle

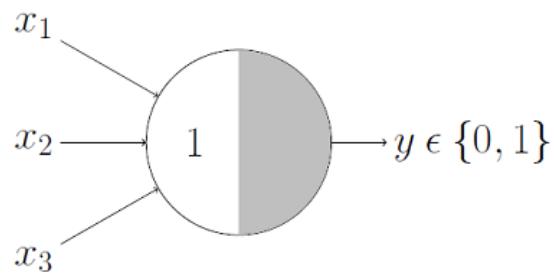


Abbildung 25: McCulloch-Pitts-Zelle - OR Gatter 2 [8]

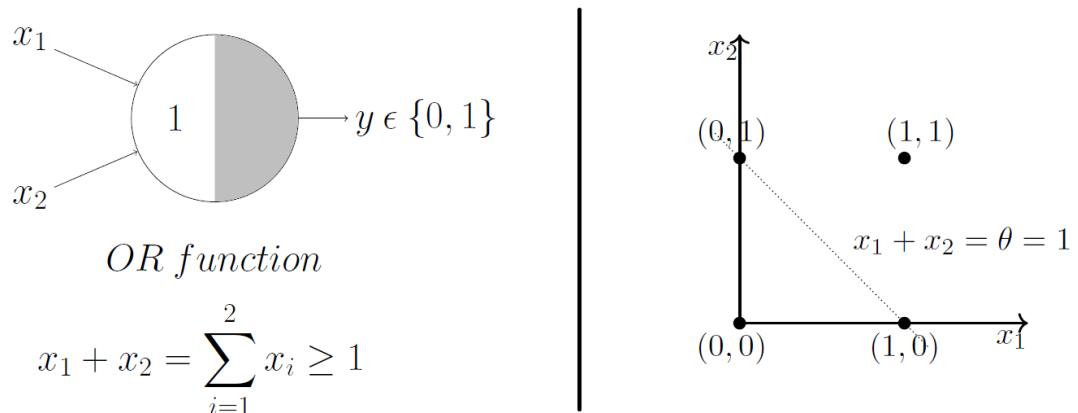


Abbildung 26: McCulloch-Pitts-Zelle - OR Gatter 3 [8]

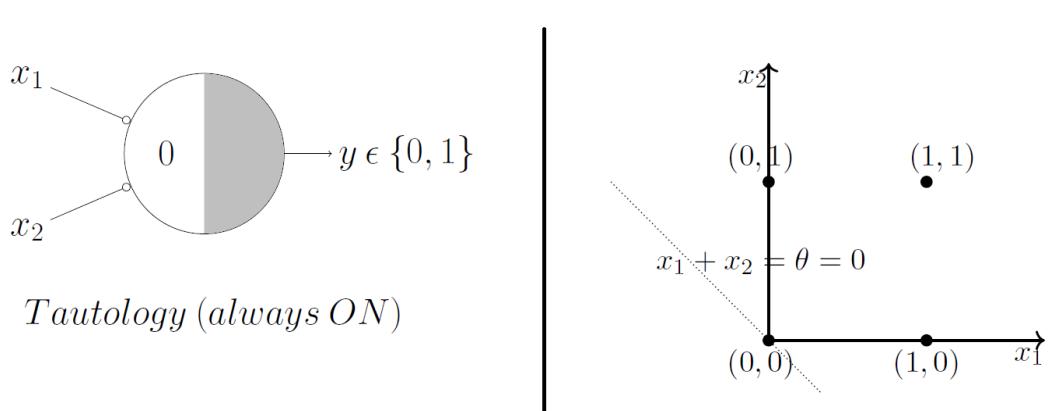


Abbildung 27: McCulloch-Pitts-Zelle - Tautologie [8]

B. Architekturen

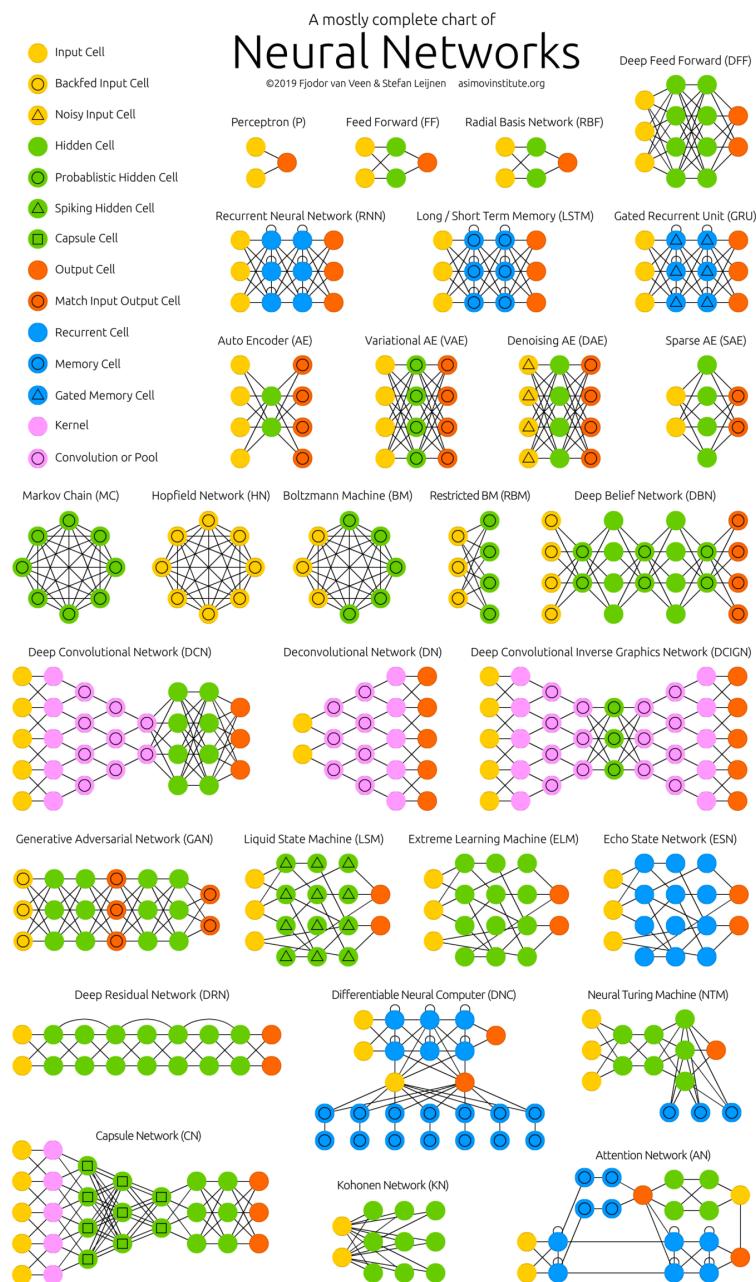


Abbildung 28: Übersicht - verschiedene Architekturen [?]