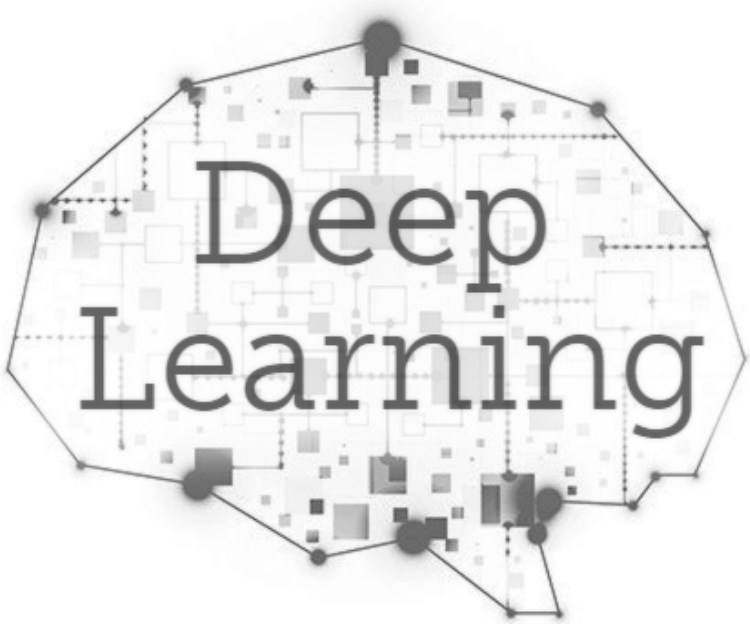


Silas Hoffmann, inf103088
5. Fachsemester
6. Verwaltungssemester

19. März 2020

Thema 2

Seminar

im Sommersemester 2020

Dozent: Prof. Dr. Dennis Säring
Fachbereich Informatik

Fachhochschule Wedel

Inhaltsverzeichnis

Abbildungsverzeichnis

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

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?

Um darauf näher eingehen zu können werden folgende Meilensteine behandelt:
todo Inhaltsverzeichnis (Sections)

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.

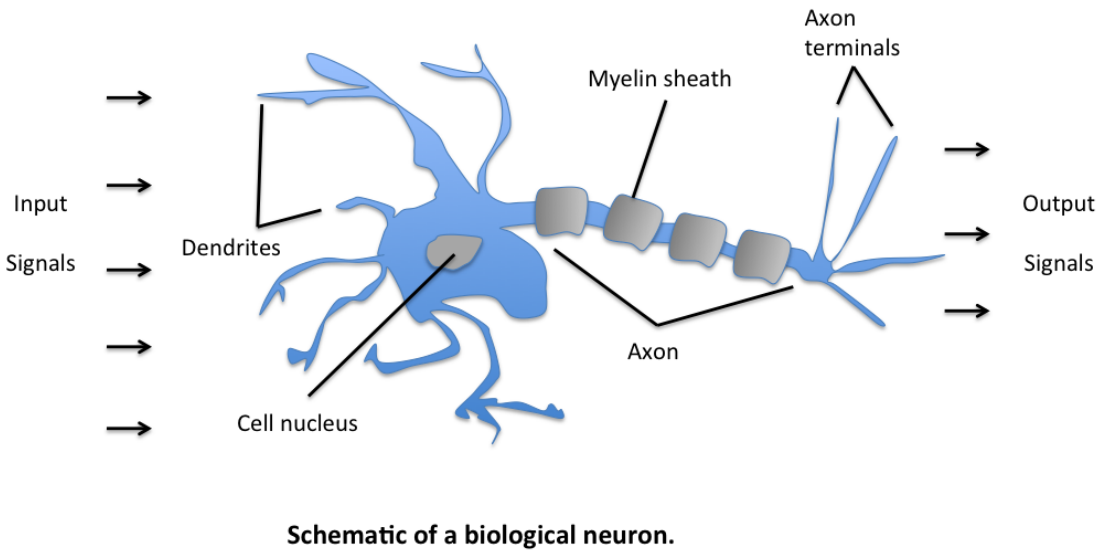
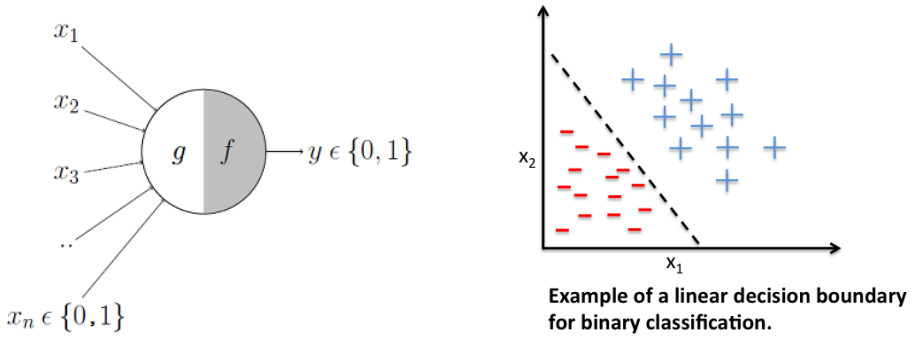


Abbildung 1: McCulloch-Pitts-Zellne - Genereller Aufbau [6]



(a) (b)

Abbildung 2: McCulloch-Pitts-Zelle: Aufbau und Klassifizierung [6]

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.

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.* [4] 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)

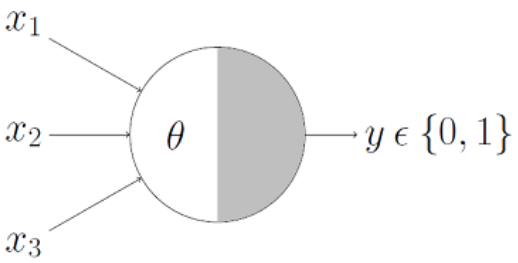


Abbildung 3: Biologische Neuronen - Notation [6]

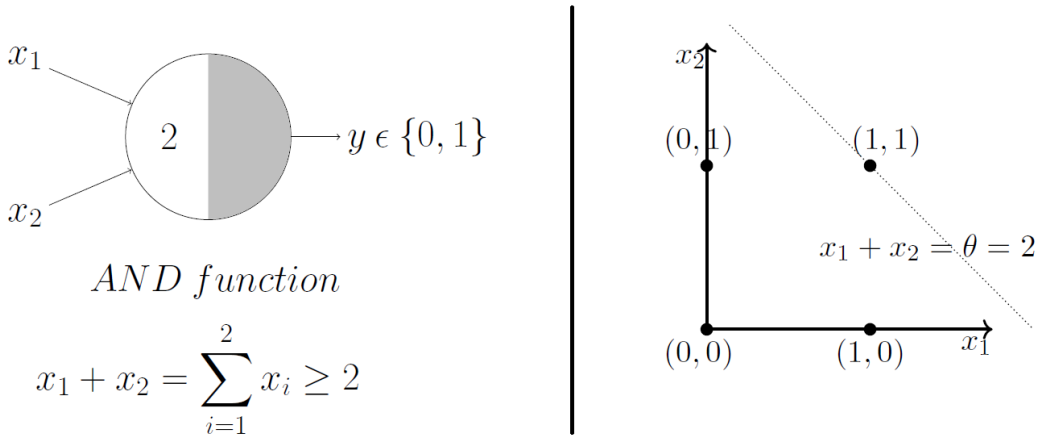


Abbildung 4: McCulloch-Pitts-Zelle - AND Gatter [6]

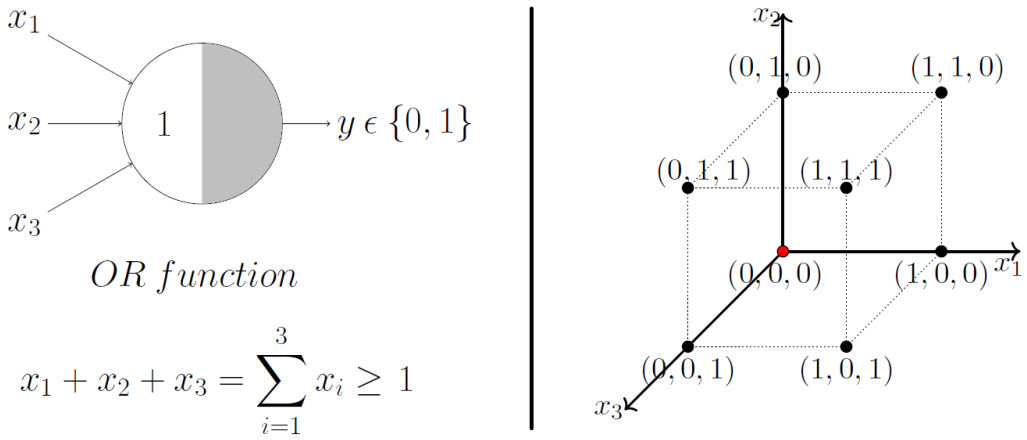


Abbildung 5: McCulloch-Pitts-Zelle - OR Gatter [6]

1.2. Nachteile bzw. Verbesserungspotenzial

- Dieses Modell erlaubt wie gesagt nur boolesche Eingabewerte, viele Modelle erfordern allerdings kontinuierliche Werte. Mit diesen wäre es zum Beispiel deutlich einfach 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 McMulloch-Pitts-Zelle.

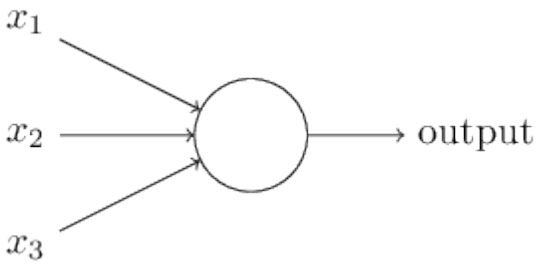


Abbildung 6: Perceptron - drei Eingabewerte [9]

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).

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

wobei gilt

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

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

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 **??**):

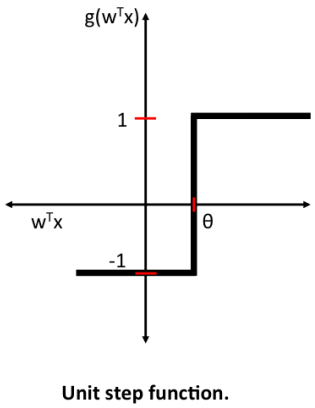


Abbildung 7: Perceptron - Einheits-Sprungfunktion [6]

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:

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

wobei gilt

$$\mathbf{z} = \mathbf{w_0 x_0} + w_1 x_1 + \cdots + w_m x_m = \sum_{j=1}^{m} x_j w_j = \mathbf{w}^T \mathbf{x} \tag{5}$$

Wichtig hierbei, es wird ein zusätzliches Gewicht welches den negativen Schwellwert hält, sowie einen zusätzlicher 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 Gleichung 6 [9]

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

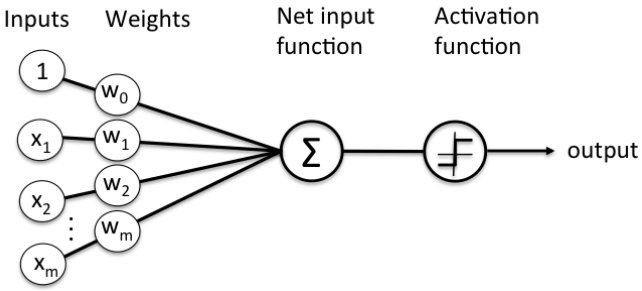


Abbildung 8: Perceptron - Modelansicht [6]

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 ein 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:

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

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 Trainingsdatensatz [1] 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:

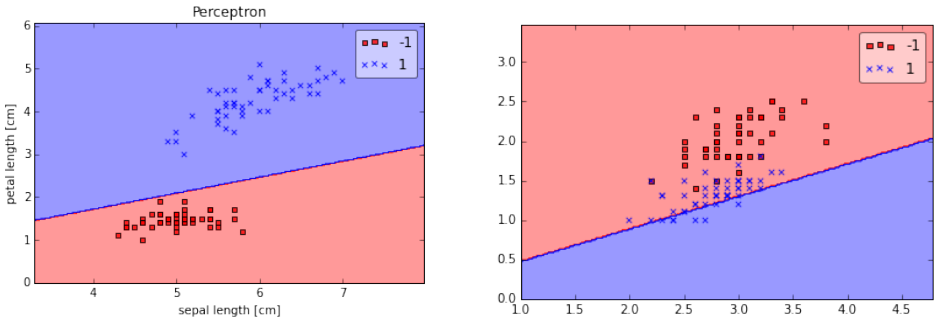
$$\begin{aligned} \Delta w_j &= \eta(1^{(i)} - -1^{(i)})\ x_j^{(i)} = \eta(2)\ x_j^{(i)} \\ \Delta w_j &= \eta(-1^{(i)} - 1^{(i)})\ x_j^{(i)} = \eta(-2)\ x_j^{(i)} \end{aligned} \tag{10}$$

Eine Implementierung des Beschriebenen Konzepts in der Programmiersprache Python ist auf Github [7] 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.

[1] Ein zweidimensionaler Datenpunkt besitzt einen zweidimensionalen Eingabevektor (also genau zwei Eingabewerte).



(a) (b)

Abbildung 9: Perceptor - Problematische Klassifizierung [6]

3. Adeline

3.1. Aufbau

Im 1959 entwickelten der Stanford Professor Bernard Widrow und der Elektroingeneur Marcian Edward Hoff das sogenannte *Adilne-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 Peceptron 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.

3.2. Lernregel

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* [2] definiert. Die Formel sieh folgendermaßen aus:

[2] engl. *sum of squared errors* (SSE)

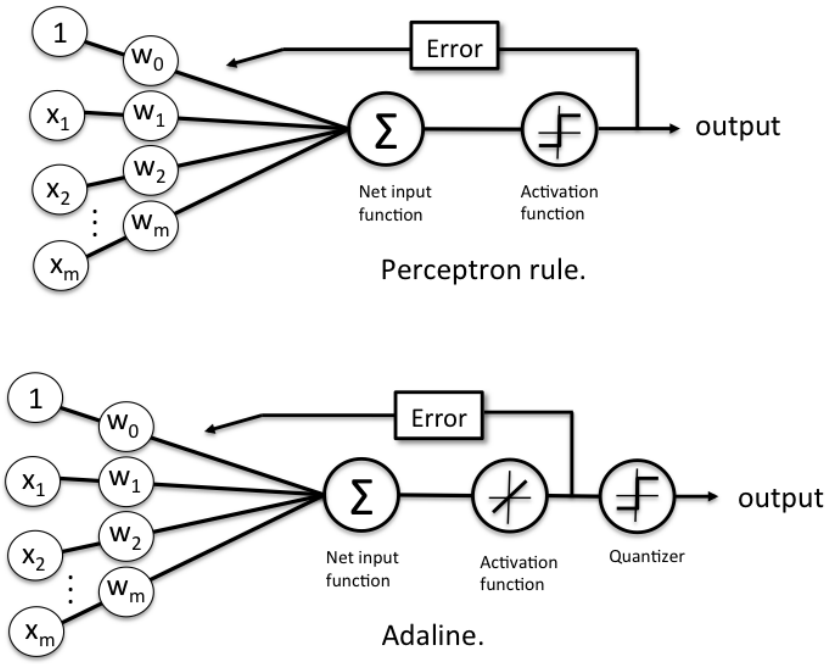


Abbildung 10: Adeline - Aufbau [6]

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

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.

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 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 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

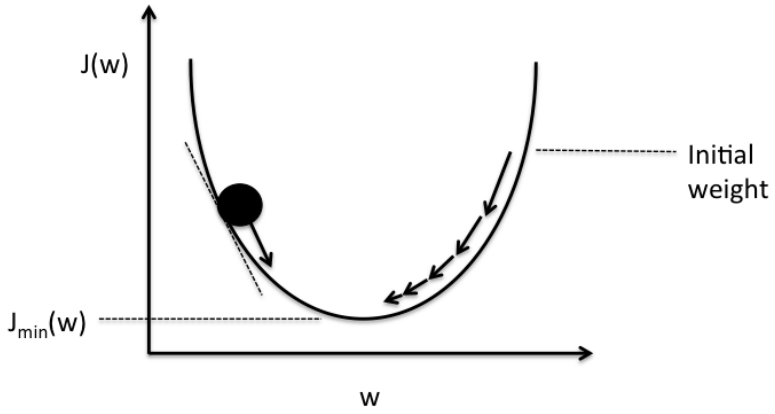


Abbildung 11: Gradientenverfahren mit eindimensionaler Kostenfunktion [6]

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.

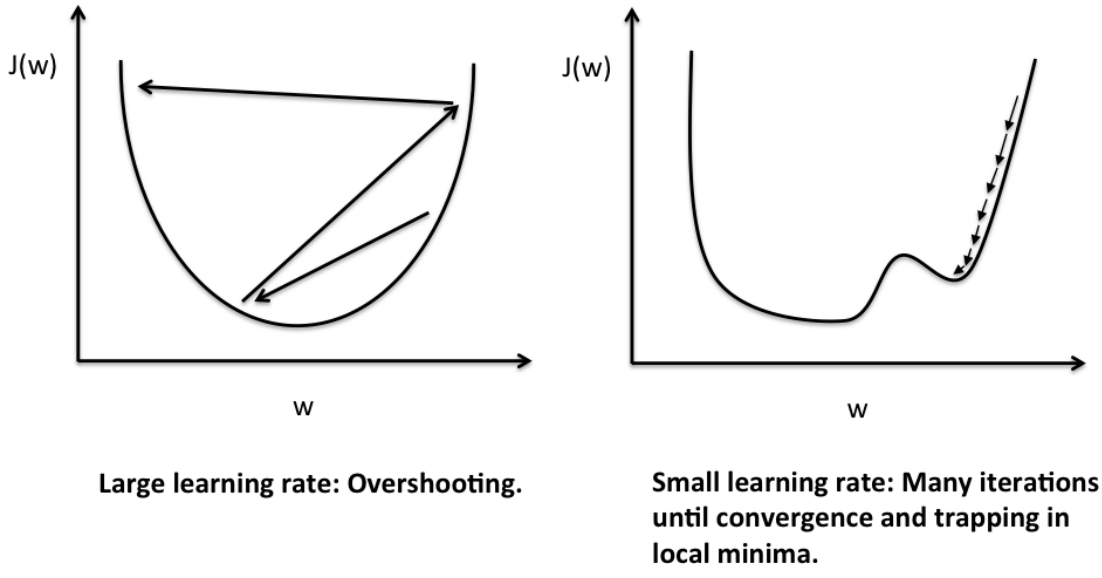


Abbildung 12: Gradientenverfahren - unterschiedliche Lernraten [6]

Letztendlich besitzt das hier betrachtete Modell aber meist mehr als ein einzelnes Ge-

wicht weswegen man sich nun damit auseinander setzten 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).

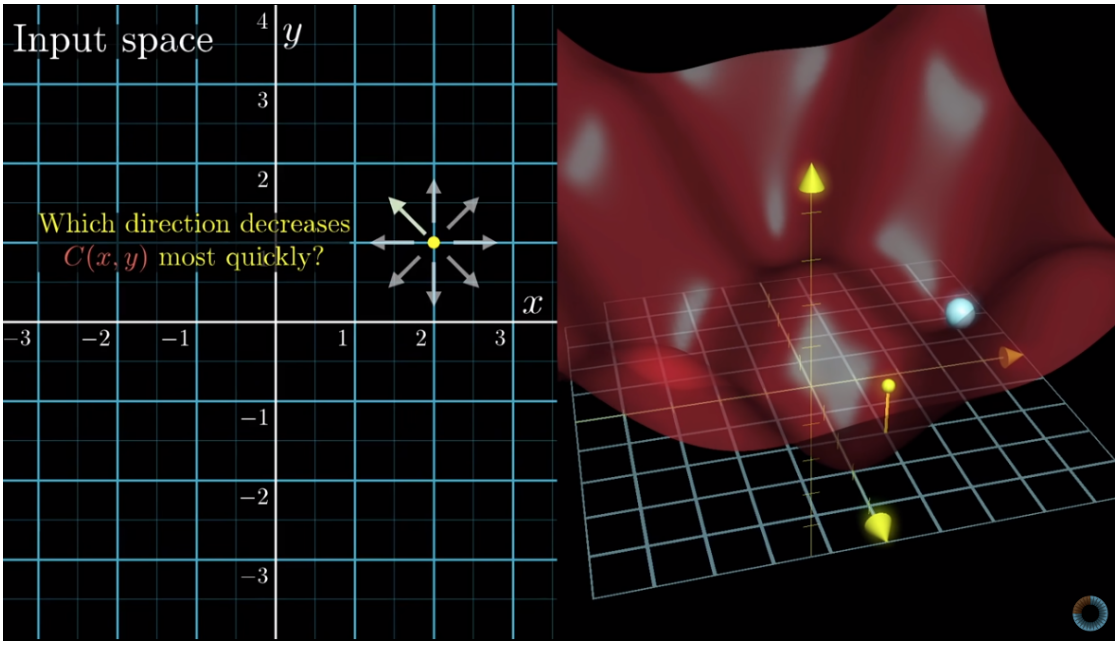


Abbildung 13: Gradientenverfahren - zweidimensionaler Eingabevektor [6]

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:

$$w = w + \Delta w \tag{12}$$

Δw stellt den angedeutete Gradientenvektor dar. Im mehrdimensionalen Raum ist diese Gewichtsveränderung allgemein Fall für den kompletten Gewichtsvektor und für den speziellen Fall mit einem einzelnen Gewicht folgendermaßen definiert.

Regel 3.1: Gradient

Allgemein:

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

Für die jeweiligen Gewichte:

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} \tag{14}$$

Exkurs - Partielle Ableitungen [5] 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).

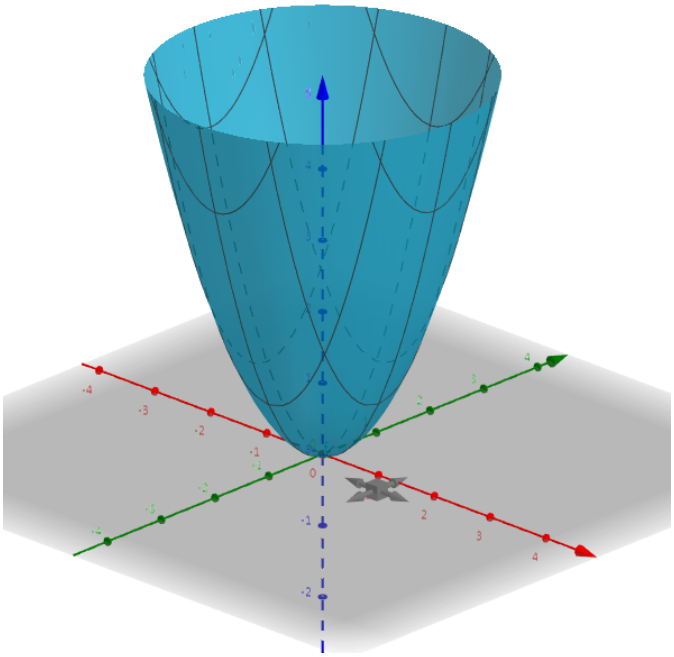
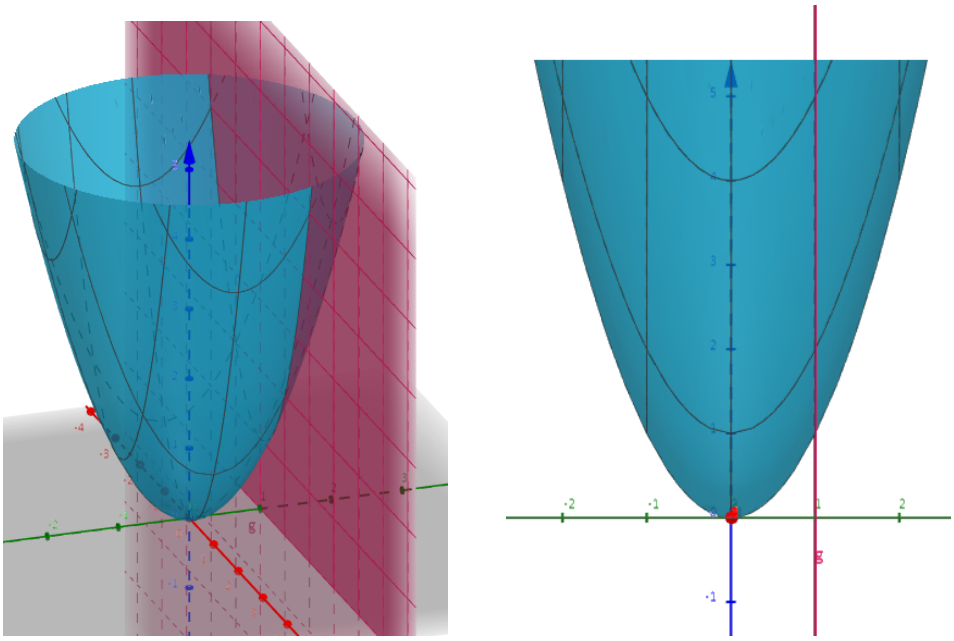


Abbildung 14: Partielle Ableitung 1 [3]

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

[3] Mit Geogebra 3d erstellt: `https://www.geogebra.org/3d?lang=de`



(a) (b)

Abbildung 15: Partielle Ableitung 2

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 *Partiale 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:

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

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

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

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

Herleitung 3.1: Gradient

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

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

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

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

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

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 Ableitungssschritt 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:

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

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

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

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

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

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 [7].

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 16 kann man erkennen wie die unterschiedlichen Arten des Gradientenabstiegs grob arbeiten. Der linke Graph beschreibt den „herkömmlichenl “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.

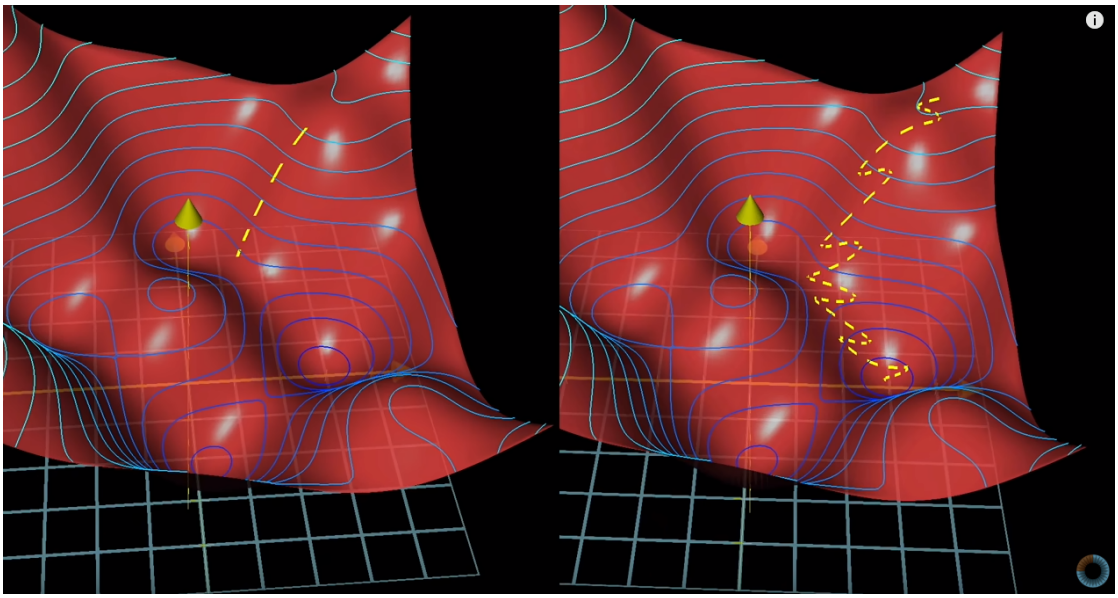


Abbildung 16: Gradientenverfahren - Vergleich [1]

4. Convolutional Neural Network

4.1. 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 “).

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

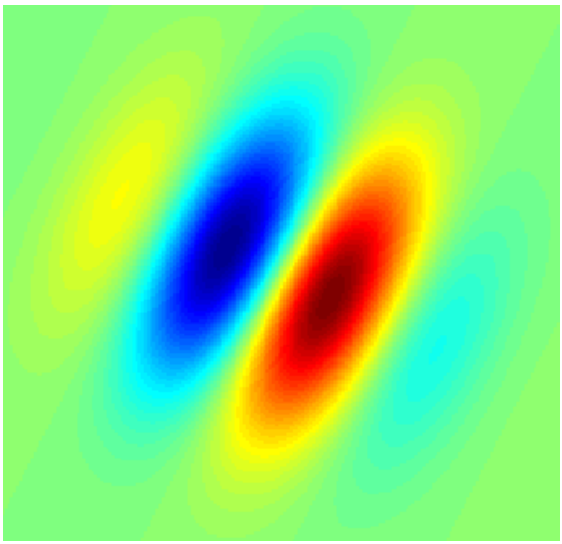


Abbildung 17: Simple Cell - Beispiel [2]

spezialisiert analysieren jedoch jeweils einen unterschiedlchen Teil des Bildes (englisch *receptive fields*, siehe Abbildung 18). So kann zum Beispiel eine komplexe Zelle horizonale 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.

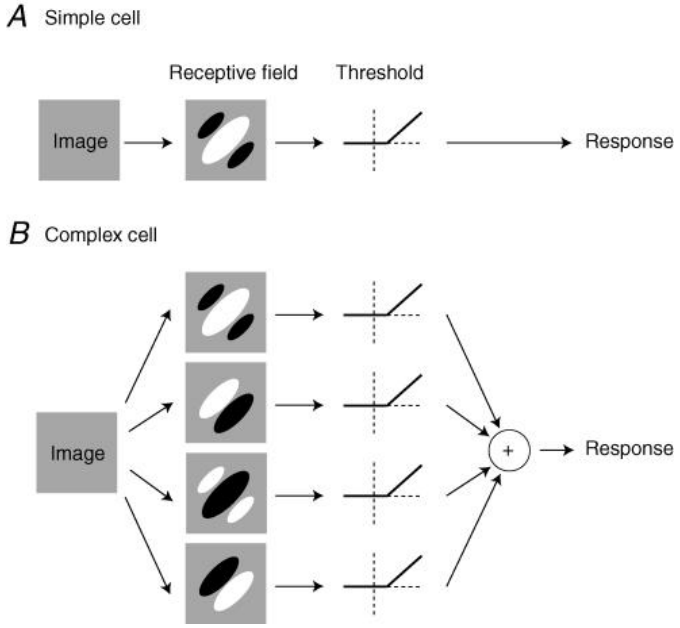


Abbildung 18: Vergleich - Simple und Complex Cell [3]

Teil III.
Aktuelle Entwicklung

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.

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] Convolutional NN). https://glassboxmedicine.com/2019/04/13/a-short-history-of-convolutional-neural-networks/. Aufgerufen am: 18-03-2020.

[3] Convolutional NN). https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1890437/. Aufgerufen am: 18-03-2020.

[4] Definition Klassifizierungssproblem. http://ekpwww.physik.uni-karlsruhe.de/~tkuhr/HauptseminarWS1112/Keck_handout.pdf. Aufgerufen am: 15-03-2020.

[5] Khan Academy - Partielle Ableitungen (Funktion mit zwei Eingabewerten). https://www.youtube.com/watch?v=1CMDS4-PKKQ&t=542s. Aufgerufen am: 16-03-2020.

[6] McCulloch-Pitts Neuron. https://towardsdatascience.com/mcculloch-pitts-model-5fdf65ac5dd1. Aufgerufen am: 14-03-2020.

[7] Perceptor - Python Implementierung. https://github.com/rasbt/mlxtend/blob/master/mlxtend/classifier/perceptron.py. Aufgerufen am: 16-03-2020.

[8] Single-Layer Neural Networks and Gradient Descent. https://sebastianraschka.com/Articles/2015_singlelayer_neurons.html. Aufgerufen am: 14-03-2020.

[9] M.A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

A. Anhang

A.1. McCulloch-Pitts-Zelle

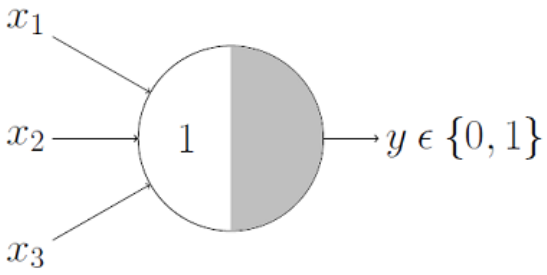


Abbildung 19: McCulloch-Pitts-Zelle - OR Gatter 2 [6]

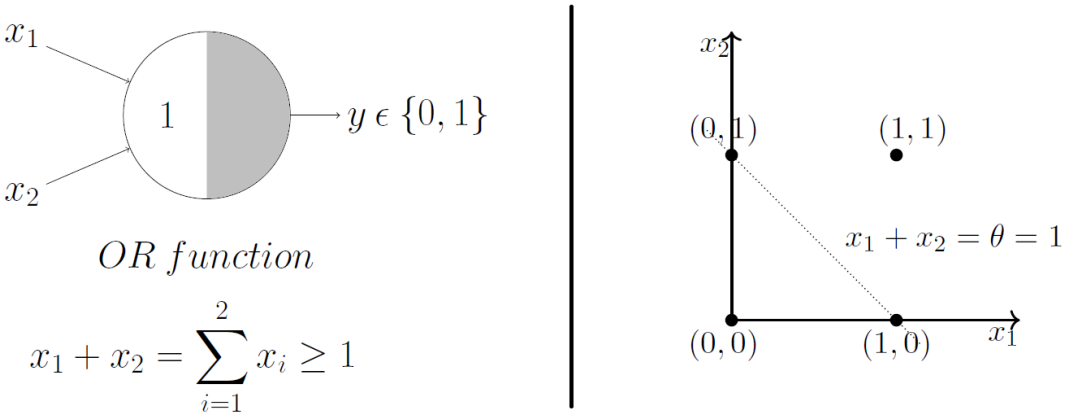


Abbildung 20: McCulloch-Pitts-Zelle - OR Gatter 3 [6]

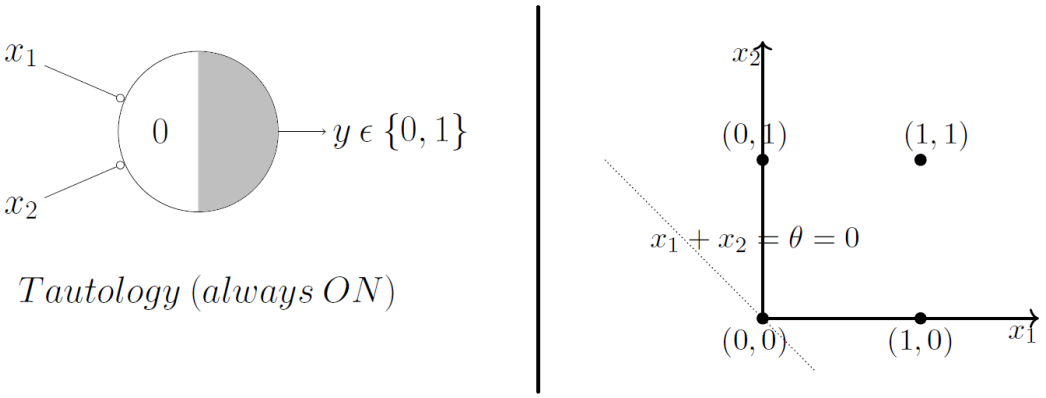


Abbildung 21: McCulloch-Pitts-Zelle - Tautologie [6]