STIFTSGYMNASIUM MELK

LEBEN LERNEN

Methoden künstlicher Intelligenz

Elias Foramitti

8A

Betreuer: Mag. Lukas Kerndler

Abt-Berthold-Dietmayr-Straße 1

3390 Melk

Schuljahr 2018/19

Abstract

Künstliche Intelligenz ist in der heutigen Zeit durch die immense Digitalisierung und die kontinuierlich steigende Personalisierung des Internets ein allgegenwärtiger Begriff und wird von vielen als die maßgebliche Technologie der Zukunft gesehen. Die Vorstöße in immer neue Gebiete, wie kürzlich etwa selbstfahrende Autos oder über weite Strecken automatisiertes Produktdesign, machen auf jeden Fall deutlich, dass Künstliche Intelligenz zukünftig zumindest eine wichtige Rolle in unserem Alltag spielen wird, sollte sich der momentane Digitalisierungstrend fortsetzen.

Trotz dieser extremen Relevanz des Themas gibt es nur relativ wenige Spezialisten und, auch wenn viele zwar eine vage Vorstellung von Künstlicher Intelligenz haben, ist das Verständnis um das konkrete Entstehen von intelligentem Verhalten aus einfachen Algorithmen und mathematischen Formeln in der breiten Bevölkerung rar.

Ziel dieser Arbeit ist es daher, zuerst den Begriff Künstlicher Intelligenz sinnvoll zu definieren und im Folgenden die herkömmlichsten oder aus anderen Gründen bezeichnenden Methoden Künstlicher Intelligenz konkret in ihrer Arbeitsweise und den daraus resultierenden Vor- und Nachteilen zu betrachten.

Inhaltsverzeichnis

1 Einleitung

1.1 Was ist Künstliche Intelligenz

Vor der Betrachtung der Arbeitsweisen Künstlicher Intelligenz ist es sinnvoll, zuerst den Begriff „Künstliche Intelligenz“ (im Folgenden auch KI) überhaupt zu definieren. Auch wenn selbst in der breiten Bevölkerung durchaus ein gewisses Grundverständnis Künstlicher Intelligenz besteht, gestaltet sich die Ausarbeitung einer eindeutig abgrenzenden Definition als recht schwierig.

Ein üblicher Ansatz ist der Vergleich mit natürlicher Intelligenz, wie etwa in der Encyclopedia Britannica:

> „*Artificial intelligence (AI), the ability of a digital computer or computer-controlled robot to perform tasks commonly associated with intelligent beings.*“ (Copeland, 1998)

Dieser Vergleich ist zwar verständlich, da in der Künstlichen Intelligenz oft sogar gezielt die Natur nachgeahmt wird, wie etwa mit neuronalen Netzen, die versuchen das menschliche Gehirn zu simulieren, jedoch ist dieser Definitionsansatz in gewisser Weise zirkulär, da nicht erklärt wird, was Intelligenz an sich bedeutet. Außerdem ist nicht undenkbar, dass Künstliche Intelligenzen einmal Aufgaben übernehmen können, die wir heute noch nicht mit intelligenten Wesen assoziieren, da diese für heute existierende Intelligenzen schlicht unlösbar sind. Zumindest wird aber schon klar, dass mit Künstlicher Intelligenz üblicherweise eine Subdisziplin der Informatik gemeint ist.

Außerdem erwähnt die oben genannte Definition Aufgaben. Also ist scheinbar ein essentieller Teil von Intelligenz die Fähigkeit Probleme zu lösen. Dazu muss die intelligente Instanz ihre Umgebung wahrnehmen und daraus auf eine Lösung des Problems schließen, um dann dementsprechend zu agieren. Dies bringt Patrick Henry Winston mit seiner Definition Künstlicher Intelligenz auf den Punkt:

> „*The study of computations that make it possible to perceive, reason, and act.*“ (Winston, 1992, S. 5)

Diese Berechnungen sind aber fast nie wie sonst in der Informatik eigentlich üblich direkt von einem Programmierer festgesetzt, sondern werden durch Algorithmen erlernt, und auf ebendiese Lern- und Verarbeitungsmethoden wird in den folgenden Kapiteln näher eingegangen.

1.2 Konkrete Umsetzung Künstlicher Intelligenz

Nach Winstons Definition nimmt eine KI also zuerst Daten von zum Beispiel Sensoren oder einer Datenbank auf, die in irgendeiner Form die Arbeitsumgebung beschreiben. Dies ist quasi die Wahrnehmung der KI (perceive). Diese Daten werden anhand von gelernten Mustern bewertet, um die Situation zu erkennen (reason). Je nach Situation gibt die KI dann wieder einen sinnvollen Output aus (act). Bei einem reinen Klassifikationsproblem wäre diese Ausgabe zum Beispiel einfach eine Beschreibung der im Schlussfolgerungsteil erkannten Situation. Die Ausgabe kann jedoch auch zum Beispiel aus Steuerungsbefehlen für Motoren bestehen. Welcher Output in welcher Situation sinnvoll ist, kann die KI natürlich nicht wissen ohne dies vorher zu erlernen und genau dieses maschinelle Lernen, also das Verknüpfen von Input mit dem richtigem Output, ist eine grundlegende Herausforderung bei der Entwicklung Künstlicher Intelligenzen.

2 Supervised Learning

Unter Supervised Learning Verfahren versteht man Methoden Künstlicher Intelligenz, die vorbereitete Trainingsdaten benötigen, aus denen sie lernen. Die Aufgabe des Lernalgorithmus ist es, so gut es geht, Rauschen aus den Daten zu filtern und eine möglichst passende Approximationsfunktion zu finden, die danach im Verarbeitungsmodus als Trenneinheit Input-Daten klassifizieren soll.

Eine typische Anwendung eines solchen Verfahrens ist zum Beispiel Objekterkennungssoftware für Bilder. Hierbei werden einem Algorithmus möglichst viele Bilder mit verschiedenen Objekten gezeigt und mitgeteilt, welches Objekt in dem Bild zu sehen ist. Er soll die Zusammenhänge der Bilddaten und der Objektart erfassen und danach fähig sein, auf unbekannten Bildern Objekte zu erkennen und deren Bezeichnung auszugeben. Jede mögliche Art von Objekt bezeichnet man als eine Klasse. Das heißt dieses Beispiel hätte so viele Klassen, wie es Arten von zu klassifizierenden Objekten gibt. (vgl. Ertel, 2016, S. 313)

2.1 Nearest Neighbour-Methode

Das wohl einfachste Lernverfahren ist die Nearest Neighbour-Methode. Hierbei werden in der Trainingsphase die Trainingsdaten ohne weitere Verarbeitung einfach abgespeichert, weshalb

man diese Methode auch als Lazy Learning Methode bezeichnet. (vgl. Pantic, 2018, S. 10) In der Anwendungsphase wird dann der ähnlichste Trainingsdatenpunkt zum zu klassifizierenden Datenpunkt gesucht. Die Ähnlichkeit lässt sich mathematisch durch den Abstand definieren. Dafür bieten sich je nach Anwendung verschiedene Abstandsmaße an. (vgl. Ertel, 2016, S. 207)

Das gängigste Abstandsmaß ist die euklidische Norm, also die Länge des Vektors zwischen zwei Punkten:

$$d(x,y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

Oft sind manche Merkmale jedoch wichtiger als andere. In diesem Fall kann man die Merkmale durch Gewichte ω skalieren:

$$d_\omega(x,y) = \sqrt{\sum_{i=1}^{n}\omega_i(x_i - y_i)^2}$$

Da man bei der Nearest Neighbour-Methode normalerweise Abstände nur vergleicht, ist die Quadratwurzel eigentlich überflüssig und wird daher meistens nicht implementiert. (vgl. Ertel, 2016, S. 207)

Bei Binärdaten bietet sich der Hamming-Abstand an. Dieser bezeichnet die Anzahl unterschiedlicher Bits zweier Datenpunkte. (vgl. Ertel, 2016, S. 210)

Bei Textklassifikation wird häufig das normierte Skalarprodukt der beiden Vektoren verwendet, das quasi deren Winkel aufeinander misst:

$$d_s(x,y) = \frac{|x| \cdot |y|}{xy}$$

Damit wären die Vektoren (1,1) und (5,5) genauso ähnlich zu einander, wie (1,1) und (2,2), da alle drei parallel aufeinander sind, also beträgt ihr normiertes Skalarprodukt jeweils 1.

Ebenfalls gebräuchlich ist der Abstand der maximalen Komponente, der nur jeweils das Merkmal betrachtet, auf dem der Unterschied der zwei verglichenen Punkte am größten ist. (vgl. Ertel, 2016, S. 245f.)

$$d_\infty(x,y) = \max_{i=1,\dots,n}|x_i - y_i|$$

In der praktischen Anwendung wird also durch alle Trainingsdaten iteriert, um den nächsten Punkt zum gegebenen Punkt zu finden. Die Klasse, der dieser nächste Nachbar angehört, wird nun auch dem neuen Datenpunkt zugeordnet.

$\text{NearestNeighbour}(M_+, M_-, s)$
$t = \underset{x \in M_+ \cup M_-}{\text{argmin}} \{d(s,x)\}$
If $t \in M_+$ Then Return(„+“)
Else Return(„-“)

Dadurch entsteht quasi eine komplexe Hypertrennfläche, die im zwei-dimensionalen Raum durch ein Voronoi-Diagramm veranschaulicht werden kann. (siehe Abbildung 1)

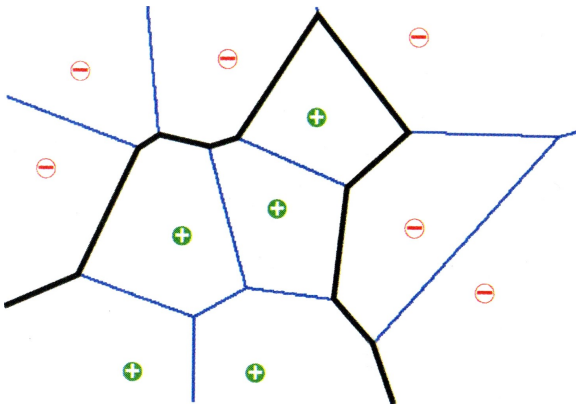


Abbildung 1: Voronoi-Diagramm

2.1.1 k-Nearest Neighbour

Eben jene Komplexität führt aber auch unweigerlich zu Überanpassung (engl. Overfitting). Dabei kommt es zu Falschklassifizierungen durch statistische Ausreißer in den Trainingsdaten. Solche Ausreißer sind bei Realdaten recht häufig. Daher ist es oft notwendig die Trennfläche zu glätten. Hierfür bietet sich die Abänderung k-Nearest Neighbour an. Bei dieser Methode wird bei der Klassifikation ein Mengenentscheid der k nächsten Nachbarpunkte getroffen. (vgl. Ertel, 2016, S. 209)

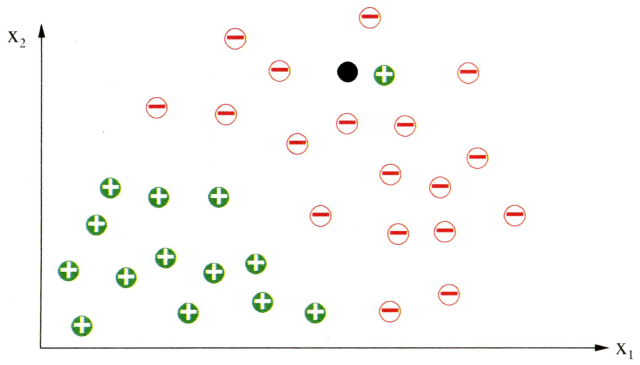


Abbildung 2: Kritischer Fall durch Ausreißer in den Daten

Oft werden die Stimmen außerdem noch zusätzlich nach ihrem Abstand zum gegebenen Punkt gewichtet, damit weiter entfernte aber stark gehäufte Nachbarn das Ergebnis nicht verfälschen. (vgl. Ertel, 2016, S. 212)

2.1.2 Vorteile und Nachteile

Man erkennt an der Beschaffenheit des Klassifikationsalgorithmus, dass mit ansteigender Trainingsdatenmenge auch die Rechenzeit bei der Klassifikation steigt. Auf der anderen Seite benötigt dafür das reine Abspeichern der Daten in der Trainingsphase deutlich weniger Aufwand als die Generalisierung, die bei Eager Learning Methoden in dieser Phase stattfindet, aber auch mehr Speicherplatz als eine generalisierte Form der Daten.

Ein großer Vorteil ist in manchen Anwendungen, dass der Algorithmus ohne große Abänderungen auch Multi-Klassen-Probleme bewältigen kann, da die zugeteilte Klasse im Grunde einfach eine arbiträre Information ist, die vorab jedem Trainingsdatenpunkt zugeteilt werden und beim Klassifizieren vom jeweils nächsten Nachbarpunkt übernommen wird. So kann man praktisch natürlich nicht, wie in dem Beispiel oben, nur „+“ und „-“ übergeben, sondern auch beliebig verschiedene Zahlen, Klassenbezeichnungen als Zeichenkette oder sogar komplexere Daten, wie ganze Objekte.

Außerdem ist die praktische Implementierung extrem einfach. Meist handelt es sich nur um wenige Zeilen Programmcode. Wenn man sich jedoch für ein falsches Abstandsmaß entscheidet oder die Gewichte der Merkmale falsch wählt, wird es zu Ergebnissen mit hoher Fehlerrate kommen. Manche komplexe Zusammenhänge können sogar gar nicht richtig bestimmt werden, da die Wichtigkeit eines Merkmals etwa von anderen Merkmalen abhängig

sein kann und nicht über die gesamte Dimension kontinuierlich verlaufen muss. Dann helfen auch Gewichte nicht weiter. (vgl. Ray, 2012)

2.1.3 Anwendung

Das Nearest Neighbour Verfahren ist immer dann effektiv, wenn es sinnvoll ist, mit ähnlichen Fällen zu vergleichen, und alle Merkmale möglichst gleich wichtig sind oder ihre Relevanz zumindest bekannt ist. Dies schließt zum Beispiel Empfehlungssysteme auf Videoplattformen (vgl. Nowling, 2016) oder medizinische Diagnosesysteme ein. (vgl. Shee / Cheruiyot / Kimani, 2014; Sarkar / Leong, 2000)

2.2 Lernen von Entscheidungsbäumen

Alle sinnvollen Lernverfahren sind zwar dazu imstande, Zusammenhänge in den Trainingsdaten zu finden, die meisten arbeiten dabei jedoch als Blackbox. Die Daten werden, wenn überhaupt, zu einer mehrdimensionalen, meist recht komplexen Funktion generalisiert und diese vollständig zu verstehen ist für Menschen üblicherweise kompliziert und zeitaufwändig.

Dieses Problem versucht das Lernen von Entscheidungsbäumen zu lösen, indem ein für Menschen einfach verständlicher Entscheidungsbaum aus den Daten generiert wird, der dann zur Klassifikation neuer Daten dient. Ein Merkmal wird hierbei immer zum Knoten und ein möglicher Wert zum Ast. (siehe Abbildung 3 und Abbildung 4) Daraus resultiert aber auch, dass diese Methode nur mit diskreten Daten arbeiten kann, da ein Knoten in einem Entscheidungsbaum nur eine endliche Zahl an Ästen haben kann.

Bei dem folgenden Beispiel soll eine KI entscheiden, ob es für den User zu gegebenen Umständen sinnvoll ist, Skifahren zu gehen. Hierzu beobachtet die KI das Verhalten eines Menschen und speichert jeweils ab, ob er an einem Tag Skifahren geht, die Entfernung zum nächsten Skigebiet mit guten Schneeverhältnissen kleiner oder größer als 100km ist, die Sonne scheint, und ob Wochenende ist. Aus den Daten soll die KI nun einen optimalen Baum erzeugen.

Variable	Werte	Beschreibung
Skifahren (Zielvariable)	ja, nein	Fahre ich los in das nächstgelegene Skigebiet mit ausreichend Schnee?
Sonne (Merkmal)	ja, nein	Sonne scheint heute?
Schnee_Entf (Merkmal)	≤ 100, > 100	Entfernung des nächsten Skigebiets mit guten Schneeverhältnissen (über/unter 100 km)
Wochenende (Merkmal)	ja, nein	Ist heute Wochenende?

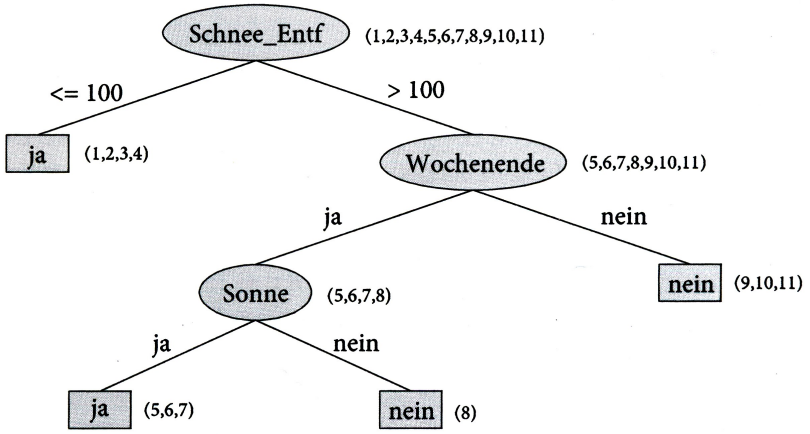


Abbildung 3: Beispielproblem Skifahren

Tag	*Schnee_Entf*	*Wochenende*	*Sonne*	*Skifahren*
1	≤ 100	ja	ja	ja
2	≤ 100	ja	ja	ja
3	≤ 100	ja	nein	ja
4	≤ 100	nein	ja	ja
5	> 100	ja	ja	ja
6	> 100	ja	ja	ja
7	> 100	ja	ja	nein
8	> 100	ja	nein	nein
9	> 100	nein	ja	nein
10	> 100	nein	ja	nein
11	> 100	nein	nein	nein

Abbildung 4: Werte Tabelle zum Beispiel aus Abbildung 3

Die einfachste Methode, einen optimalen, also möglichst fehlerfrei klassifizierenden und möglichst kompakten, Entscheidungsbaum zu finden, wäre alle möglichen Entscheidungsbäume für das Problem zu generieren und den besten auszuwählen. Jedoch steigt so die Rechenzeit exponentiell mit wachsender Zahl an Dimensionen der Trainingsdaten. Sinnvoller ist also den Entscheidungsbaum mit einer heuristischen Methode von oben nach unten aufzubauen. (vgl. Ertel, 2016, S. 217-219)

2.2.1 Entropie als Maß des Informationsgehalts

Mit dieser Top-Down Methode wird immer das Attribut, das den höchsten Informationsgewinn verspricht, ausgewählt. Der Informationsgehalt eines Merkmals lässt sich durch die Entropie als Maß für die Unsicherheit der Wahrscheinlichkeitsverteilung der Daten eines Merkmals beschreiben. Diese ist wie folgt definiert:

$$H(p) = H(p_1, ..., p_n) := -\sum_{i=1}^{n} p_i \log_2(p_i)$$

Für eine vertiefende Beschäftigung mit dieser Formel wären in Shannon, 1948, S. 10 und S. 28f. ausführlichere Erklärungen und eine Herleitung der Formel zu finden.

$H(p)$ ist hierbei die Entropie und p ein Vektor, der alle Wahrscheinlichkeiten des jeweiligen Auftretens eines möglichen Werts eines Merkmals enthält.

Die Wahrscheinlichkeiten $p_1, ..., p_n$ werden durch das jeweilige Auftreten des Werts in den Trainingsdaten abgeschätzt. Auf die Ergebnismenge Skifahren aus dem obigen Beispiel angewendet:

$$D = (\text{ja}, \text{ja}, \text{ja}, \text{ja}, \text{ja}, \text{ja}, \text{nein}, \text{nein}, \text{nein}, \text{nein}, \text{nein})$$

$$p_1 = P(\text{ja}) = \frac{6}{11} \text{ und } p_2 = P(\text{nein}) = \frac{5}{11}$$

$$H(D) = -(\frac{6}{11}\log_2(\frac{6}{11}) + \frac{5}{11}log_2(\frac{5}{11})) \approx 0.994$$

Also beträgt die Entropie, also der Informationsgehalt, von Skifahren ungefähr 0.994.

Mithilfe der Entropie lässt sich nun der Informationsgewinn berechnen:

$$G(D, A) = H(D) - \sum_{i=1}^{n} \frac{|D_i|}{|D|} H(D_i)$$

Die Datenmenge D wird dabei durch das n-wertige Attribut A in $D_1 \cup D_2 \cup ... D_n = D$ aufgeteilt. Die Herleitung dieser Formel ist in Ertel, 2016, S. 221f. zu finden.

Angewendet auf das Ski-Beispiel:

$$G(D, \text{Schnee\_Entf}) = H(D) - (\frac{4}{11}H(D_{\leq 100}) + \frac{7}{11}H(D_{>100})) \approx 0,994 - (\frac{4}{11} \cdot 0 + \frac{7}{11} \cdot 0,863) \approx 0,445$$

Die Ergebnismenge Skifahren wird also geteilt in Skifahren, wenn Schnee_Entf $\leqslant 100$, und Skifahren, wenn Schnee_Entf > 100 .

Diese Formel wird nun auf alle Attribute angewendet, das Attribut mit höchstem Informationsgewinn wird ausgesucht und als erster Knoten angenommen. Dies wird für die daraus entstehenden Teildatenmengen und die überbleibenden Attribute rekursiv weitergeführt, solange bis ein Ast zu einem sicheren Ergebnis kommt, das heißt der Informationsgewinn für alle verbleibenden Attribute 0 ist, oder keine Attribute mehr verbleiben. (vgl. Ertel, 2016, S. 223)

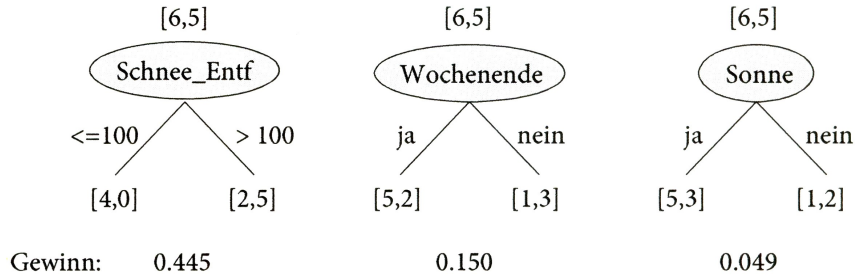


Abbildung 5: Der Informationsgewinn im Beispiel aus Abbildung 3 und 4

2.2.2 Vorteile und Nachteile

Der größte Vorteil dieser Methode ist, wie oben schon genannt, die einfache Verständlichkeit für den Menschen. So kann der entstandene Baum auch später noch angepasst werden, während das bei üblichen generalisierten Funktionen eher schwierig ist.

Dafür können Entscheidungsbäume nur mit diskreten Daten arbeiten. Der bekannte C4.5 Algorithmus kann zwar auch mit reellen Daten umgehen, teilt diese jedoch einfach in Intervalle, um wieder diskrete Attributwerte zu erhalten. (vgl. Quinlan, 1996, S. 85) Diese Vorgangsweise wurde auch im oben angeführten Beispiel für das Merkmal Schnee_Entf angewendet, indem diese eigentlich reellen Werte in ≤100 und >100 geteilt wurden.

Außerdem neigen Entscheidungsbäume stark zu Überanpassung. Dies wird meist durch Pruning, also „Stutzen“, der Bäume gelöst, indem zu lange Äste abgeschnitten werden. Ebenfalls gängig sind Entscheidungswälder, die aus mehreren geeigneten Entscheidungsbäumen bestehen, die mit Mengenentscheiden klassifizieren. (vgl. Tong / Xie / Hong / Fang / Shi / Perkins / Petricoin, 2004) Dies erschwert jedoch wieder die Verständlichkeit für den Menschen. (vgl. Ertel, 2016, S. 231)

2.2.3 Anwendung

Maschinell-gelernte Entscheidungsbäume werden überall dort eingesetzt, wo es darum geht, Wissen aus Daten zu extrahieren und dann nicht nur anwendbar, sondern auch dem Menschen zugänglich zu machen, zum Beispiel bei Konsumentenanalyse, Finanzbetrugserkennung, Fehlersuche in technischen Geräten oder Software und Management. (vgl. Lee / Chrysostomou / Chen / Liu, 2008)

2.3 One-Class Learning

In vielen Anwendungen stehen einem nur Trainingsdaten einer Klasse zur Verfügung. Um zum Beispiel eine KI zu trainieren, die erkennen soll ob eine Industriemaschine defekt ist, möchte man normalerweise nicht extra einen Defekt nachahmen, sondern die KI soll an der fehlerfrei laufenden Maschine lernen, wie dieser zu beobachtende Fall aussieht und den defekten Zustand nur im Umkehrschluss erkennen. (vgl. Ertel, 2016, S. 242)

Die üblichste Methode ist die Nearest Neighbour Data Description, eine Variation des schon bekannten Nearest Neighbour Verfahrens. Hierbei werden alle Trainingsdaten in Lazy-Learning-Manier nur normalisiert, also auf allen Dimensionen linear auf das Intervall [0,1] skaliert, und ansonsten unverarbeitet abgespeichert. Die Normalisierung ist wichtig, damit alle Merkmale gleich gewertet werden, egal welchem Wertebereich sie entspringen. In der Klassifizierungsphase wird nun der nächste Nachbar in den Trainingsdaten zum unbekannten Punkt gesucht und deren Abstand mit dem durchschnittlichen Abstand nächster Nachbarn innerhalb der Trainingsdaten verglichen.

„*Gegeben sei eine Trainingsdatenmenge* $X = (x_1, \ldots, x_n)$ *bestehend aus n Merkmalvektoren. Ein neu zu klassifizierender Punkt q wird akzeptiert, wenn*

$$D(q, NN(q)) \leq \gamma \overline{D}$$

das heißt, wenn der Abstand zu einem nächsten Nachbarn nicht größer ist als $\gamma \overline{D}$. *Hierbei ist* $D(x, y)$ *eine Metrik, zum Beispiel wie hier verwendet die Euklidische Norm. Die Funktion*

$$NN(q) = \underset{x \in X}{\operatorname{argmin}} \{D(q, x)\}$$

ermittelt einen nächsten Nachbarn von ***q*** *und*

$$\overline{D} = \frac{1}{n} \sum_{i=1}^{n} D(x_i, NN(x_i))$$

ist der mittlere Abstand der nächsten Nachbarn zu allen Datenpunkten in X .“ (Ertel, 2016, S.243)

γ ist in dieser Formel ein Parameter zur Einstellung der Empfindlichkeit, dieser wird üblicherweise mittels Kreuzvalidierung so bestimmt, dass die Fehlerrate bei der Klassifikation möglichst gering ist. Eine genauere Erklärung der Kreuzvalidierung ist in Ertel, 2016, S. 232-234 nachzulesen.

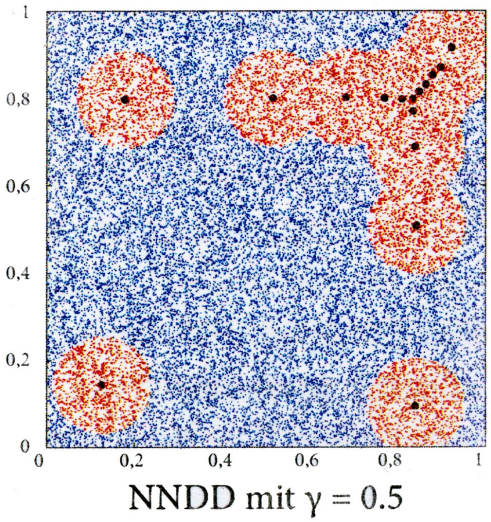


Abbildung 6: NNDD-Verfahren

2.3.1 Vorteile und Nachteile

Die Fehlerrate dieser Methode ist im Vergleich zu anderen Lernverfahren besonders bei wenigen Trainingsdaten extrem hoch. Daher versucht man normalerweise, wenn irgendwie möglich, Trainingsdaten für alle Klassen zu sammeln.

Wenn dies jedoch nicht möglich ist, vor allem bei Prozessen, die nur unter hohen Kosten verändert oder gestoppt werden können, um andere Daten zu sammeln, wie etwa in vielen Bereichen der Industrie, bietet die Methode eine sinnvolle Alternative. (vgl. Ertel, 2016, S. 242f.)

2.4 Perzeptron

Das Perzeptron ist eines der einfachsten, aber dadurch auch ein recht eingeschränktes Verfahren.

Formal nimmt ein Perzeptron einen Inputvektor x auf und bildet aus diesem und einem Gewichtsvektor w das Skalarprodukt. Es wird also jede Koordinate des Inputvektors je mit einem zugehörigen Gewicht multipliziert und folgend werden die Produkte addiert. Eine

Inputkoordinate wird jedoch immer auf den Wert 1 festgesetzt. Das zugehörige Gewicht, der sogenannte „Bias“, fungiert dadurch als additive Konstante. Das Skalarprodukt wird dann weiter durch die Aktivierungsfunktion φ verarbeitet, die dann die jeweilige Klasse ausgeben soll. Für φ kommen hierbei verschiedene Funktionen in Frage. Für die einfache einlagige Verwendung ist eine Schwellenwertfunktion meist die beste Wahl, da sie nur 0 und 1 als eindeutige Klassenzuordnung ausgeben kann:

$$\Theta(x) = \begin{cases} 0, & \text{falls } x < 0 \\ 1, & \text{sonst} \end{cases}$$

In komplexeren Anwendungen werden Perzeptronen jedoch üblicherweise zu mehrlagigen Konstrukten, sogenannten „neuronalen Netzen“, verbunden. Da der Lernalgorithmus von neuronalen Netzen eine Ableitung von φ benötigt und Schwellenwertfunktionen nicht differenzierbar sind, wird hierbei daher oft die Sigmoidfunktion verwendet, die stetige Werte zwischen 0 und 1 ausgibt:

$$S(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{1+e^x}$$

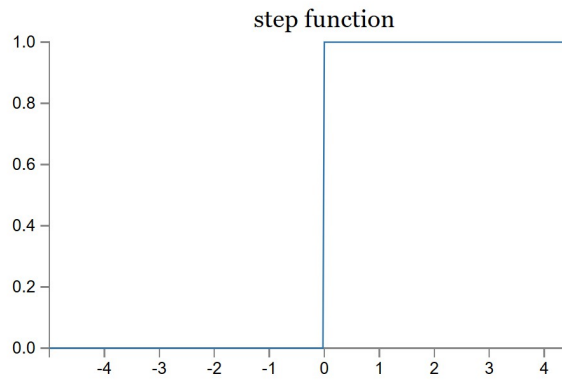


Abbildung 7: Schwellenwertfunktion

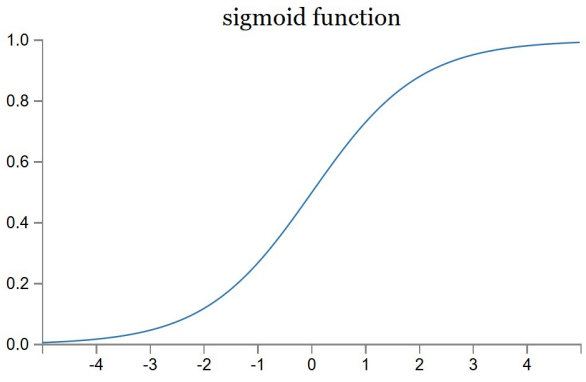


Abbildung 8: Sigmoidfunktion

Das gesamte Perzeptron kann also mathematisch durch folgende Formel beschrieben werden:

$$o = \varphi(x \cdot w) = \varphi(\sum_{i=1}^{n} x_i w_i)$$

(vgl. Otte, 2009, S. 2)

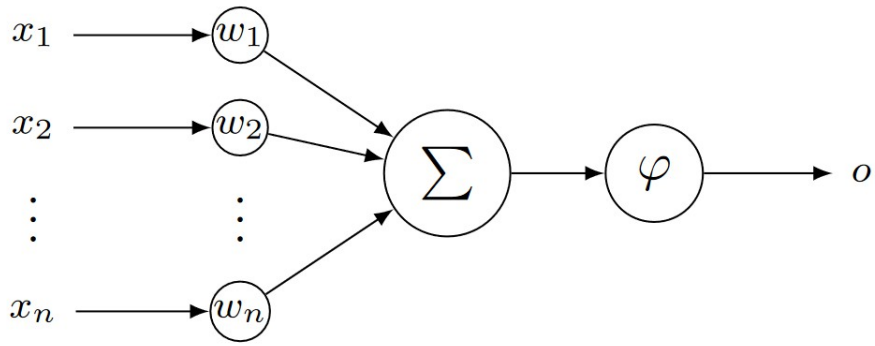


Abbildung 9: Funktionsweise eines Perzeptrons

Geometrisch kann man ein einlagiges Schwellenwert-Perzeptron als eine (n-1)-dimensionale Hyperebene im n-dimensionalen Datenraum beschreiben. Alle Daten werden danach klassifiziert, auf welcher Seite der erzeugten Hyperebene die Datenpunkte liegen. Im zwei-dimensionalen Raum beispielsweise wäre ein Perzeptron also als Trenngerade darzustellen. Dadurch ist ein einlagiges Perzeptron jedoch nur auf linear separable, zwei-klassige Datenmengen anwendbar, was leider nur eher selten der Fall ist. Im Gegenzug weist es sowohl beim Lernen als auch beim Klassifizieren einen äußerst geringen Rechenaufwand auf. (vgl. Ertel, 2016, S. 199-201)

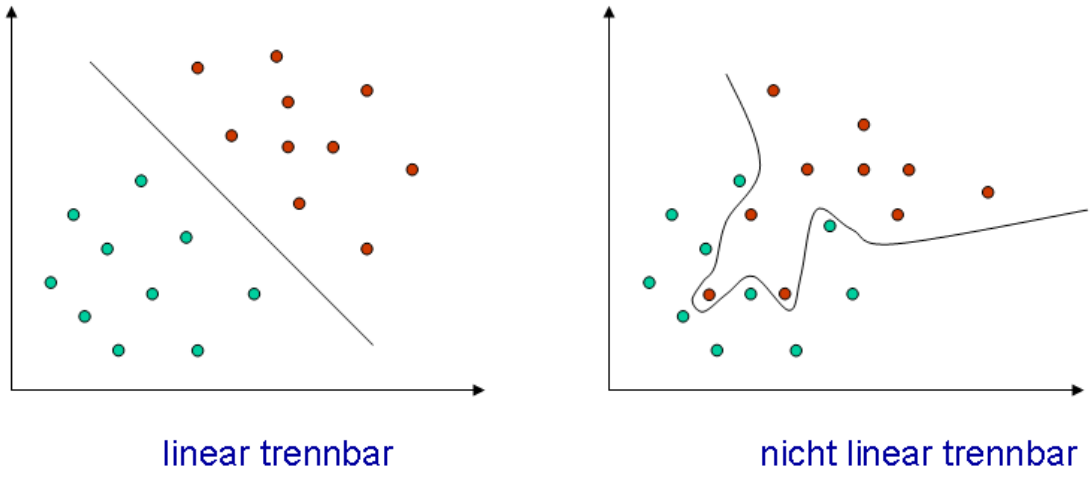


Abbildung 10: Linear separierbare und nicht linear separierbare Mengen

Wie bereits erwähnt, können Perzeptronen auch zu mehrlagigen und weit mächtigeren neuronalen Netzen verbunden werden. Dies war eigentlich schon von seinem Entwickler Frank Rosenblatt so gedacht, der das Perzeptron auf Basis des Neuronen-Modells von McCulloch und Pitt konzipierte. (vgl. Müller, 2017)

2.4.1 Trainieren eines einlagigen Schwellenwert-Perzeptrons

Das Trainieren eines einfachen Perzeptrons hat das Ziel, den Gewichtsvektor, also die Gewichte und den Bias, so anzupassen, dass zumindest alle Trainingsdaten korrekt klassifiziert werden. Dazu wird der Gewichtsvektor zuerst mit zufälligen Werten angenommen. Dann iteriert man durch alle Trainingsdaten, lässt diese vom Perzeptron klassifizieren und für alle falsch klassifizierten Inputvektoren werden diese je nach tatsächlicher Klasse des Trainingsdatenpunktes zum Gewichtsvektor addiert beziehungsweise subtrahiert. Für welche Klasse man dabei addiert oder subtrahiert entscheidet nur die Bezeichnung der Klasse durch das Perzeptron als Ausgabewert 0 oder 1. Dies wird solange wiederholt bis kein Trainingsdatenpunkt mehr falsch klassifiziert wird. Unbedingt zu beachten ist außerdem beim Trainieren, dass auch für alle Trainingsdaten immer eine bestimmte Koordinate durchgängig als 1 festgesetzt werden muss, damit auch der Bias trainiert wird und Inputvektor und Gewichtsvektor die gleiche Dimension haben. (vgl. Ertel, 2016, S. 202-204)

PerzeptronLernen(M_+, M_-)
  $w =$ beliebiger Vektor reeller Zahlen
  Repeat
    For all $x \in M_+$
      If $wx \leqslant 0$ **Then** $w = w + x$
    For all $x \in M_-$
      If $wx > 0$ **Then** $w = w - x$
  Until alle $x \in M_+ \cup M_-$ werden korrekt klassifiziert
  Return(w)

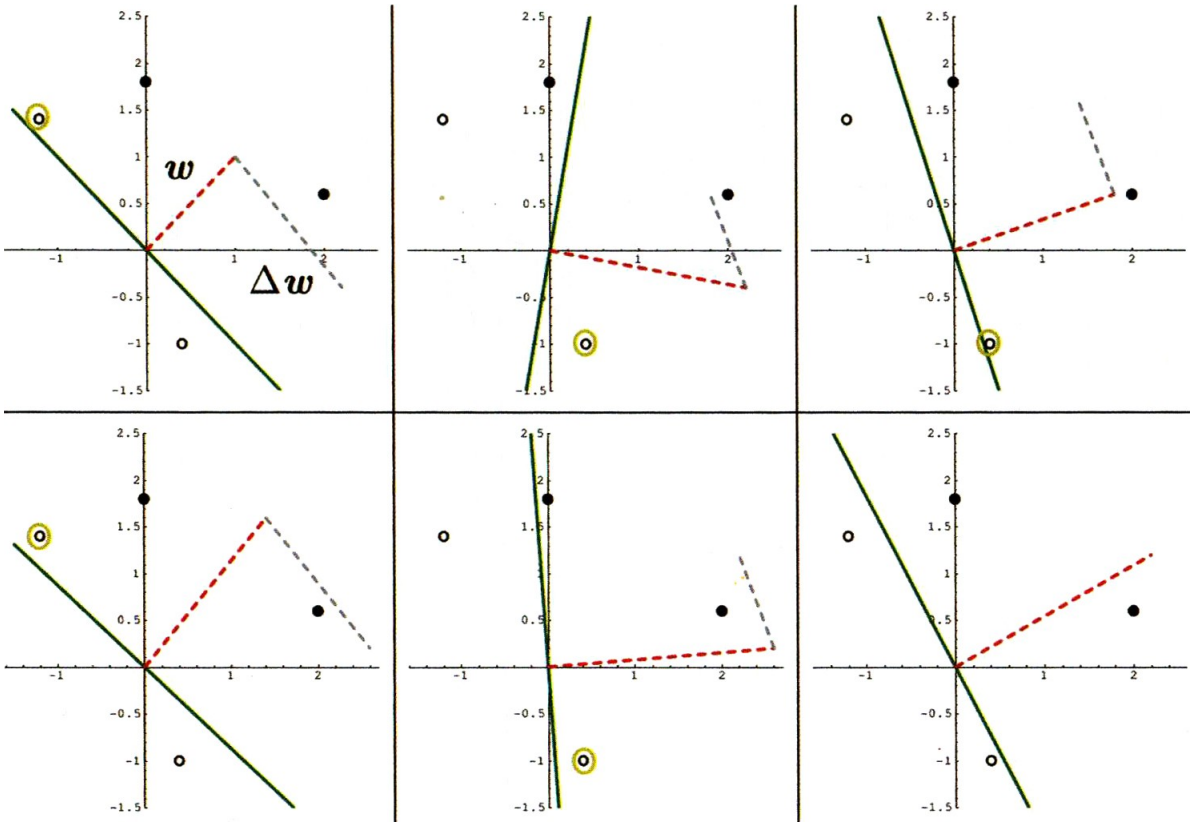


Abbildung 11: Geometrische Veranschaulichung des Lernalgorithmus eines Perzeptrons

In Abbildung 11 erkennt man, dass der Gewichtsvektor quasi ein Normalvektor der Trenngerade ist und das Addieren beziehungsweise Subtrahieren eines falsch klassifizierten Punktes den Gewichtsvektor zu beziehungsweise weg von dem jeweiligen Punkt lenkt, sodass die Trenngerade genau zwischen die beiden Klassen konvergiert. Ein Bias wurde bei diesem Beispiel nicht trainiert. Dieser würde jedoch durch die Addition beziehungsweise Subtraktion von 1 die Funktion für jeden falsch klassifizierten weißen Punkt nach oben und für jeden schwarzen nach unten verschieben.

Da diese Methode oft unnötig viele Iterationen benötigt, wird sie oft durch eine Lernrate erweitert. Der Inputvektor wird mit dieser multipliziert, bevor er zum Gewichtsvektor addiert oder subtrahiert wird, wodurch er mehr oder weniger Einfluss auf den Gewichtsvektor hat. Die Lernrate muss dabei je nach Problemstellung festgelegt werden, meist ist sie aber kleiner als 1, verringert also den Einfluss auf den Gewichtsvektor. (vgl. Müller, 2017)

2.4.2 Vorteile und Nachteile

Das Perzeptron ist ein Lernverfahren aus den frühen Anfängen der Entwicklung Künstlicher Intelligenz. Dadurch ist es in seiner Funktionsweise auch recht eingeschränkt. So kann es nur mit zwei Klassen umgehen, die auch noch linear separierbar sein müssen. Daher ist es heute in dieser ursprünglichen Form in der Praxis kaum noch in Verwendung. Der grundlegende Aufbau wird jedoch bis heute in den Neuronen von Neuronalen Netzen weiterverwendet, die momentan wohl das mächtigste und vielversprechendste Supervised Learning Verfahren darstellen. (vgl. Müller, 2017)

2.5 Neuronale Netze

Neuronale Netze verbinden nun mehrere Perzeptronen miteinander, wobei diese in diesem Kontext üblicherweise Neuronen genannt werden.

2.5.1 Feedforward

Ein Neuron nimmt dabei als Input die Ausgabewerte anderer Neuronen und verarbeitet sie entsprechend dem Perzeptronenschema. Die eingespeisten Werte werden also nach dem jeweiligen Gewichtsvektor des Neurons gewichtet, aufaddiert und durch die Aktivierungsfunktion zu einem Ausgabewert verarbeitet. Als Aktivierungsfunktion ist vor allem die Sigmoidfunktion üblich, die kontinuierliche Werte von 0 bis 1 ausgibt und daher differenzierbar ist, was sich für den im folgenden Kapitel 2.5.2 beschriebenen

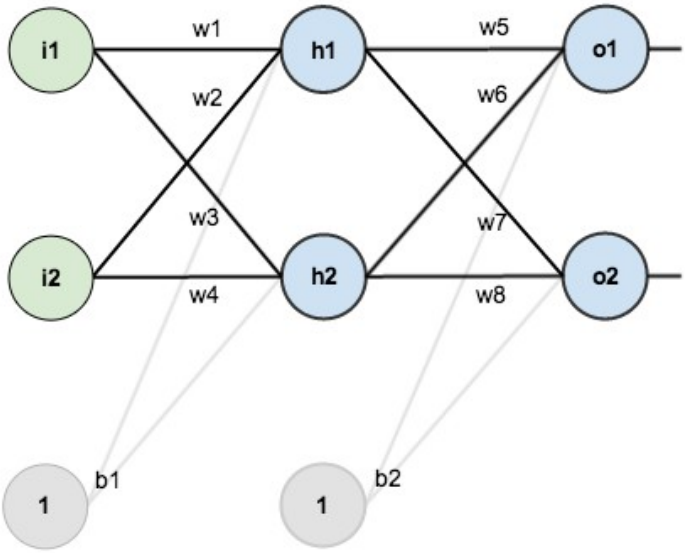


Abbildung 12: Struktur eines Neuronalen Netzes mit veranschaulichten Bias-Neuronen

Backpropagationalgorithmus als essentiell herausstellen wird. Wie ein Perzeptron hat ein Neuron auch einen Bias, der genauso auf einer eigenen Koordinate des Gewichtsvektors liegt, die konstant immer mit 1 multipliziert wird. Es ist also so als hätte jede Schicht außer der Ausgabeschicht ein zusätzliches Bias-Neuron, das konstant den Wert 1 ausgibt.

Die meisten neuronalen Netze sind in Schichten aufgebaut. Jedes Neuron nimmt also alle Ausgabewerte der letzten Schicht als Input. Die erste Schicht ist dabei die Inputschicht, mit der Daten in das neuronale Netz eingespeist werden, und die letzte die Outputschicht, deren Ausgabe die Klasse der Eingabedaten darstellen soll. Die Ausgabeschicht kann natürlich auch aus mehreren Neuronen bestehen, wodurch neuronale Netze auch bestens für Multi-Klassen-Probleme geeignet sind. Durch die Eigenschaft der Sigmoidfunktion kontinuierliche Werte auszugeben, nimmt auch jedes Outputneuron Werte zwischen 0 und 1 an. Diese können als Wahrscheinlichkeiten für die Zugehörigkeit der Input-Daten zu der jeweiligen Klasse verstanden werden. Zwischen Input- und Outputschicht liegen üblicherweise noch mehrere versteckte Schichten. Diese Vorgehensweise des Weitergebens von Schicht zu Schicht wird als Feedforward bezeichnet. (vgl. Loy, 2014)

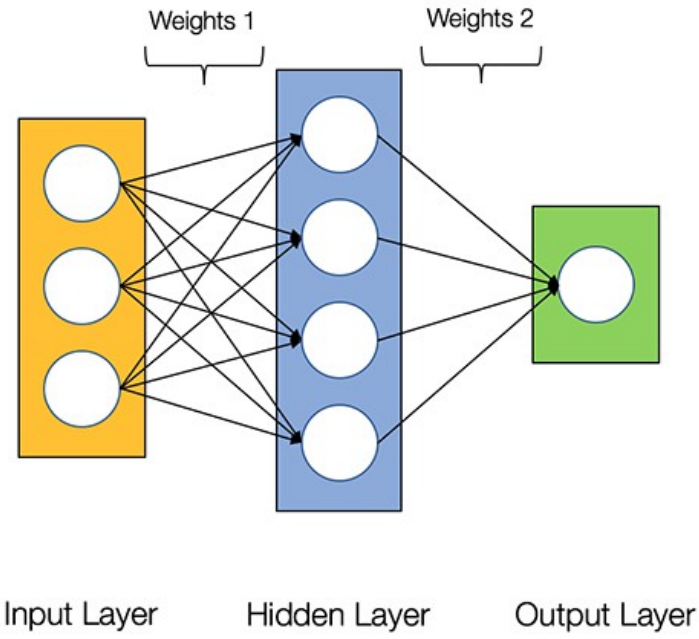


Abbildung 13: Struktur eines Neuronalen Netzes

Ein typisches Beispiel wäre etwa die Erkennung von handschriftlichen Ziffern. Input wäre dabei das gescannte Bild der Ziffer. Der Grauwert jedes Pixels würde nun in ein eigenes Inputneuron gespeist werden. Diese Werte werden dann durch mehrere Schichten und in jedem Neuron durch einen eigenen Gewichtsvektor verarbeitet. Die Outputschicht müsste aus je einem Neuron pro möglicher Ziffer, also insgesamt zehn Neuronen, bestehen. Wenn nun das fünfte Neuron der Ausgabeschicht bei einem Bild zum Beispiel 0.8 ausgibt, dann ist sich das neuronale Netz zu 80% sicher, dass das Inputbild die Ziffer 5 darstellt.

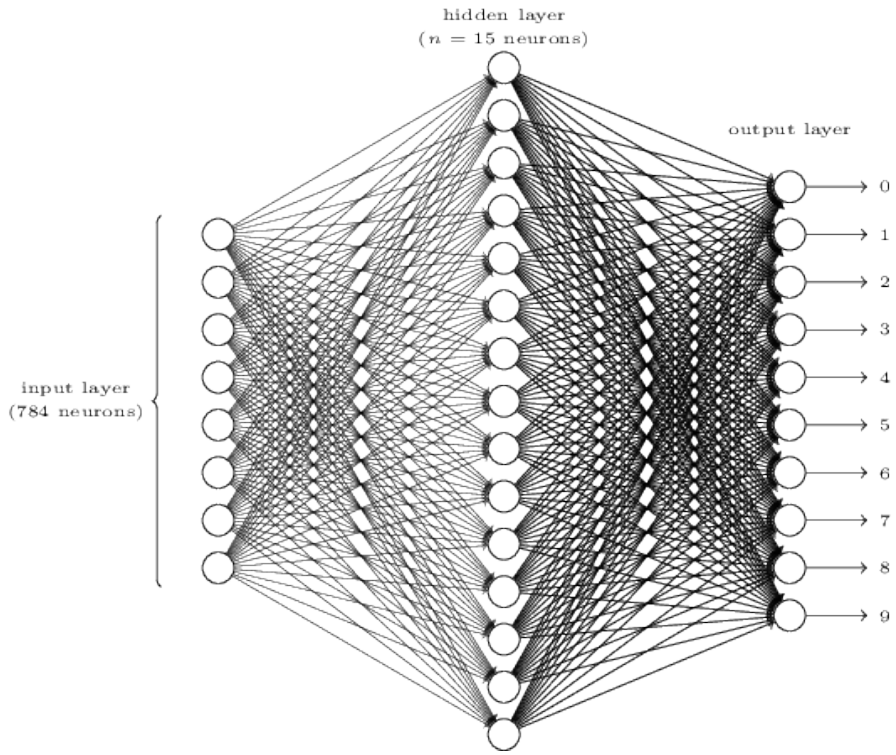


Abbildung 14: Mögliche Struktur eines Neuronalen Netzes zur Ziffernerkennung auf einem Bild von 28x28 Pixel

2.5.2 Trainieren eines neuronalen Netzes durch Backpropagation

Die Qualität einer Klassifikation eines neuronalen Netzes ist abhängig von dessen Gewichten. Beim Trainieren möchte man also alle Gewichte so anpassen, dass sie zumindest die Trainingsdaten richtig klassifizieren. Es werden zuerst wieder zufällige Gewichte angenommen und dann nacheinander alle Trainingsdaten in das neuronale Netz gespeist und die Fehler in den Gewichten jeweils ermittelt und ausgebessert.

Dazu benötigt man zuerst eine Funktion, die die Qualität der Klassifikation misst, eine sogenannte Fehlerfunktion:

$$\sigma(y_1, o_1) = \sum_{b=1}^{n_1} (y_{1,b} - o_{1,b})^2$$

wobei $y_{1,b}$ der Soll-Ausgabewert, $o_{1,b}$ der tatsächliche Ausgabewert des b. Ausgabeneurons und n_1 die Anzahl der Ausgabeneuronen ist. (Das System der Indizes wird im Folgenden noch näher erläutert.) Es werden also die Differenzen von Soll- und Ist-Wert aller Ausgabeneuronen quadriert, um absolute Werte zu erhalten, und aufaddiert. Diese Formel gibt

natürlich nur den Fehler einer einzelnen Klassifikation an. Der Fehler aller Trainingsbeispiele wäre der Durchschnitt aller Einzelfehler.

Für diese Funktion soll nun auf ein Minimum gefunden werden, indem die Gewichte abgeändert werden. Dazu muss zuerst der Einfluss aller Gewichte auf das Ergebnis ermittelt werden. Mit diesem Wissen können dann alle Gewichte entsprechend angepasst werden. Die effizienteste Methode dafür ist das Gradientenabstiegsverfahren. Dabei wird für die Fehlerfunktion die Ableitung nach den Gewichten berechnet, da diese die Steigung der Funktion angibt und damit auch die Richtung, in der die Funktion am stärksten abfällt, also den Weg zum nächsten lokalen Minimum. Anders ausgedrückt ist die Ableitung einer Funktion nach einer darin enthaltenen Variable ein Maß dafür, wie sich der Funktionswert bei einer minimalen Änderung der Variable ändern würde. Die Ableitung der Fehlerfunktion nach einem Gewicht gibt also an, welche Auswirkung eine kleine Änderung des Gewichts auf den Gesamtfehler hat. Mit diesem Wissen können dann alle Gewichte so angepasst werden, dass der Fehler des Neuronalen Netzes zumindest beim momentanen Trainingsbeispiel sinkt. Einziges Problem hierbei ist, dass σ nicht direkt von den Gewichten w, sondern von $y_{1,b}$ und $o_{1,b}$ abhängig zu sein scheint. Es ist jedoch zu bedenken, dass $y_{1,b}$ bekannt ist und $o_{1,b}$ nur eine komplexe Funktion abhängig vom ebenfalls bekannten Outputvektor der vorherigen Schicht o_2 und dem neuroneneigenen Gewichtsvektor ist. Tatsächlich ist σ also nur von den Gewichten abhängig. (vgl. Loy, 2014)

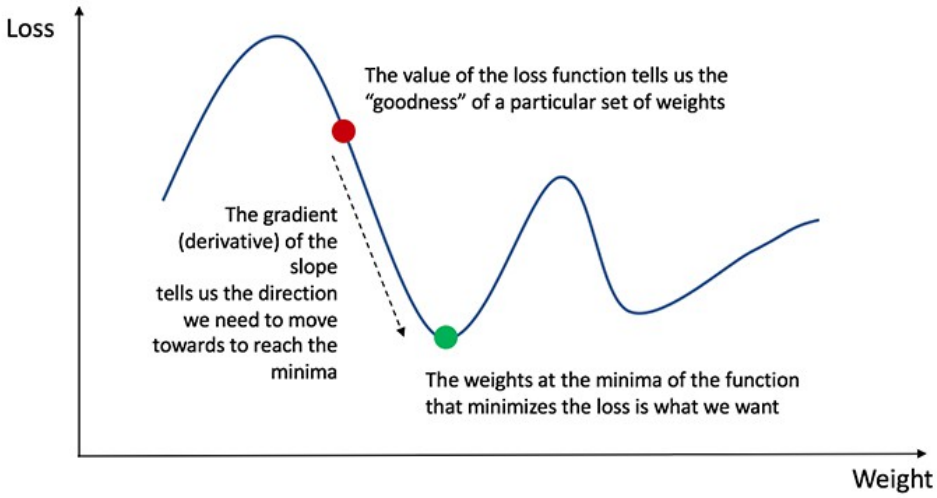


Abbildung 15: Prinzip des Gradientenabstiegs

Man berechnet also gemäß der Kettenregel zuerst die Auswirkung der Neuronen-Outputs auf die Fehlerfunktion also die Ableitung von σ nach $o_{1,b}$:

$$\sigma'(o_{1,b}) = 2(y_{1,b} - o_{1,b})$$

Dann folgt die Ableitung des Outputs $o_{1,b}=\varphi(o_2 w_{1,b})$ nach dem Skalarprodukt $o_2 w_{1,b}$:

$$\frac{\delta o_{1,b}}{\delta o_2 \cdot w_{1,b}} = \varphi'(o_2 \cdot w_{1,b})$$

Schließlich bleibt noch die Ableitung des Skalarprodukts $o_2 \cdot w_{1,b} = \sum_{c=1}^{n_2} o_{2,c} w_{1,b,c}$ nach dem jeweiligen Gewicht $w_{1,b,c}$:

$$\frac{\delta o_2 \cdot w_{1,b}}{\delta w_{1,b,c}} = o_{2,c}$$

Nach der Kettenregel beziehungsweise, da die Auswirkung eines Gewichts in der letzten Schicht auf den Gesamtfehler immer das Produkt aus den Auswirkungen der Zwischenfunktionen ist, ergibt sich so die Ableitung der Fehlerfunktion σ nach einem Gewicht $w_{1,b,c}$:

$$\sigma'(w_{1,b,c}) = 2(y_{1,b} - o_{1,b})\varphi'(o_2 \cdot w_{1,b}) o_{2,c}$$

Die erste Indexzahl steht hierbei immer für die Schicht, wobei beginnend von der Ausgabeschicht, also vom Ende des Netzwerks, mit 1 zu zählen begonnen wird, die zweite für das Neuron dieser Schicht und die dritte für das Gewicht. Also ist $w_{a,b,c}$ das Gewicht c des Neurons b auf der Schicht a, das den Ausgabewert des Neurons c der vorherigen Schicht gewichtet. $y_{1,b}$ ist demnach der Sollwert des Ausgabeneurons b, $o_{a,b}$ ist der Ausgabewert des Neurons b der Schicht a, o_a ist der Ausgabevektor der Schicht a, $w_{a,b}$ ist der Gewichtsvektor des Neurons b der Schicht a und n_a ist die Zahl der Neuronen der Schicht a.

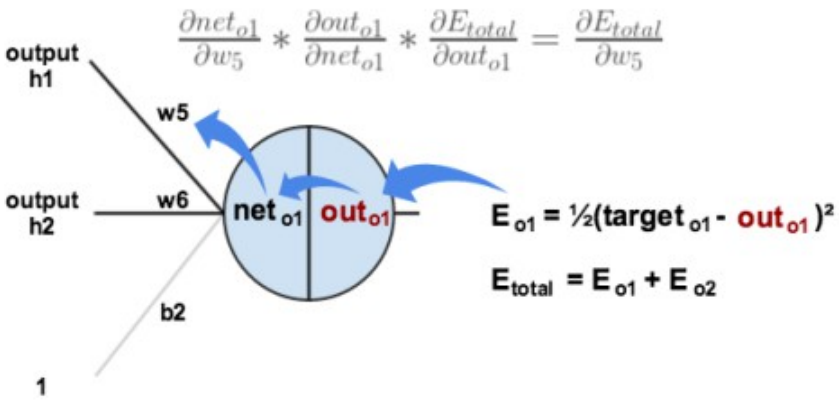


Abbildung 16: Das Backpropagation-Prinzip in der letzten Schicht

Nach dem gleichen System kann man nun fortfahren und die Auswirkung der Neuronenoutputs in der vorletzten Schicht auf die Fehlerfunktion berechnen, wobei hier zu beachten ist, dass dieses Neuron über mehrere Wege Einfluss auf den Fehler hat, weshalb die Ableitungen aller dieser Wege aufaddiert werden müssen:

$$\sigma'(o_{2,b}) = \sum_{i=1}^{n_1} \sigma'(o_{1,i})\varphi'(o_2 \cdot w_{1,i})w_{1,i,b}$$

Daraus können dann wieder die Gradienten der jeweiligen Gewichte bestimmt werden. Nach diesem Muster wird das gesamte Neuronale Netz von hinten nach vorne durchgearbeitet: Man berechnet die Gradienten aller Neuronen-Outputs einer Schicht und davon ausgehend dann die Gradienten aller Gewichte und wiederum der Neuronen-Outputs der vorherigen Schicht.

Allgemein mathematisch formuliert ergibt sich also:

$$\sigma'(w_{a,b,c}) = \sigma'(o_{a,b})\varphi'(o_{a+1} \cdot w_{a,b})o_{a+1,c}$$

Die Ableitung der Kostenfunktion nach der Ausgabe eines Neurons $\sigma'(o_{a,b})$ lässt sich rekursiv beschreiben:

$$\sigma'(o_{a,b}) = \sum_{i=1}^{n_{a-1}} \sigma'(o_{a-1,i})\varphi'(o_a \cdot w_{a-1,i})w_{a-1,i,b}, \text{ wobei } \sigma'(o_{1,b}) = 2(y_{1,b} - o_{1,b})$$

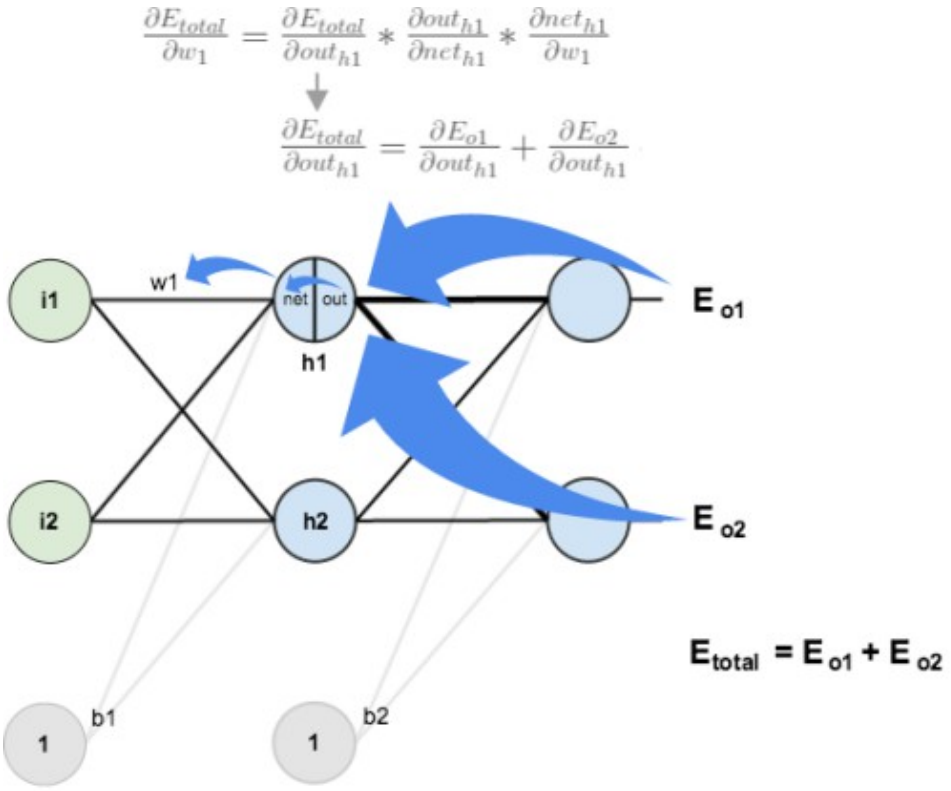


Abbildung 17: Ausweitung des Backpropagation-Prinzips auf vorige Schichten

Mit dieser Formel kann nun für jedes Gewicht eines herkömmlichen mehrlagigen Perzeptrons dessen Gradient berechnet werden. Dieser ist dann zu diesem Gewicht zu addieren, um ein neues, besser angepasstes Gewicht zu bilden. Oft werden die Gradienten vor der Addition ähnlich wie beim einlagigen Perzeptron jedoch noch mit einer konstanten Lernrate multipliziert. (vgl. Mazur, 2015)

Am effizientesten funktioniert Backpropagation, wenn man die Gradienten der Gewichte je für mehrere Beispiele, sogenannte Mini-Batches, berechnet und die Gewichte erst durch deren Durchschnittswerte updatet. Dies nennt man auch stochastisches Gradientenabstiegsverfahren. (vgl. Nielsen, 2018b)

2.5.3 Bildverarbeitung durch Convolutional Layers

Das Problem herkömmlicher mehrlagiger Perzeptronen ist, dass mit jedem zusätzlichen Input-Neuron die Zahl der Gewichte und damit der Rechenaufwand stark ansteigt. Besonders bei Bildern, die Millionen von Pixeln mit meist jeweils drei Farbwerten haben können, bedeutet, dass vor allem im Backpropagationprozess einen immensen Rechenaufwand. Als Lösung für

dieses Problem werden bei Anwendungen Neuronaler Netze mit Bildern, wie zum Beispiel bei Objekterkennung in Fotos, sogenannte Convolutional Layers eingefügt. Diese soll Merkmale im Bild finden, welche meistens in den ersten Schichten einfache Konzepte, wie Kanten und Ecken, sind und mit aufsteigender Schichtzahl immer komplexer werden können, wie zum Beispiel geometrische Formen oder sogar ganze Objekte. (vgl. Karpathy, o.J.)

Der Input einer Convolutional Layer muss in Form einer Matrix sein, wie zum Beispiel die Grauwertmatrix eines Fotos. Auf diese Matrix wird ein Filter angewendet. Dieser Filter ist selbst eine kleinere Matrix, die Pixel für Pixel über die Inputmatrix geschoben wird, das Skalarprodukt der beiden Matrizen berechnet und in einer neuen Matrix, der sogenannten „Feature Map“, ausgibt. Um zu verhindern, dass Daten am Rand des Bildes verloren gehen, wird die Inputmatrix manchmal an den Rändern durch Zellen mit dem Wert 0 erweitert. Dies nennt man auch Zero-Padding.

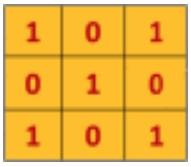


Abbildung 18: Beispiel einer Filtermatrix

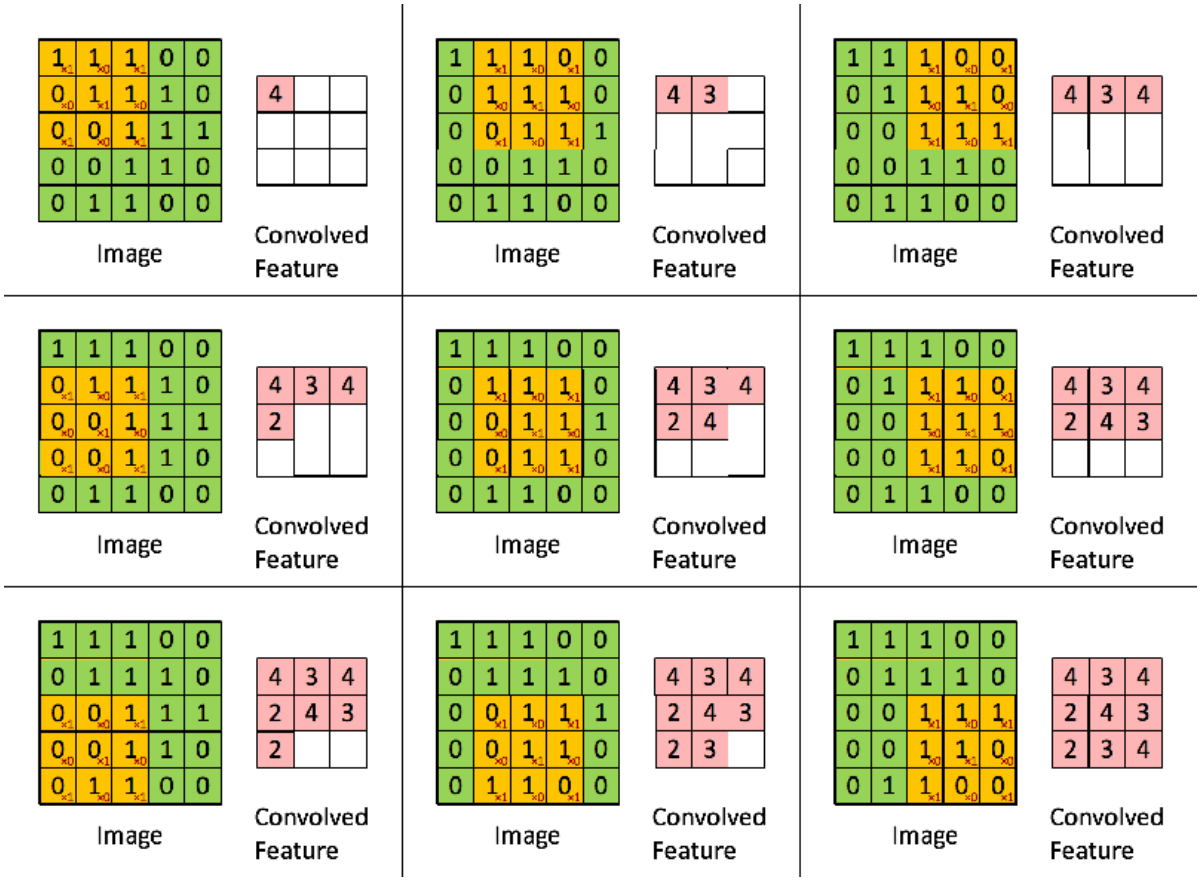


Abbildung 19: Anwendung des Filters aus Abbildung 18 auf eine Matrix ohne Zero-Padding

Meistens wird auf jeden Wert der Feature Map noch eine nicht-lineare Funktion, wie Sigmoid oder Rectifier, angewendet. Dies entspricht der Aktivierungsfunktion eines Perzeptrons. Ursprünglich war die Sigmoidfunktion in Neuronalen Netzen am üblichsten. In den letzten Jahren erkannte man jedoch, dass Rectifier vor allem bei Convolutional Neuronal Networks zu einem besseren Ergebnis führt. Ein Rectifier ist folgend definiert:

$$f(x) = \max(0, x)$$

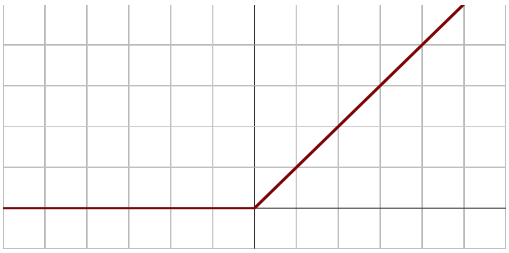


Abbildung 20: Rectifier

Die Größe der Filtermatrix wird zwar vorab festgelegt, die Werte in der Matrix müssen jedoch, wie die Gewichte in einem mehrlagigen Perzeptron gelernt werden. Da eine Convolutional Layer im Grunde einer herkömmlichen Perzeptronenschicht entspricht, in der eben jedes Neuron nur mit einigen wenigen Neuronen der Vorgängerschicht verbunden ist und alle Neuronen denselben Gewichtsvektor beziehungsweise Filter verwenden, kann der Filter genauso durch Backpropagation gelernt werden.

Oft werden außerdem noch Pooling Layers eingeführt, um die Matrix zu verkleinern. Pooling Layers teilen die Matrix dafür in Unterbereiche ein und geben je nach Art der Pooling Layer den Maximal- oder Durchschnittswert oder die Summe der Werte als einen neuen Matrixwert aus. Dadurch wird jeder Bereich zu einem einzigen Pixel zusammengefasst. Max Pooling Layers haben sich dabei in der Praxis als beste Option erwiesen. In der Backpropagation wird bei selektiven Funktionen wie dem Maximalwert immer nur der Pfad zurückberechnet, der jeweils gewählt wurde, da natürlich alle anderen keine Auswirkung auf das Endergebnis und damit die Fehlerfunktion hatten.

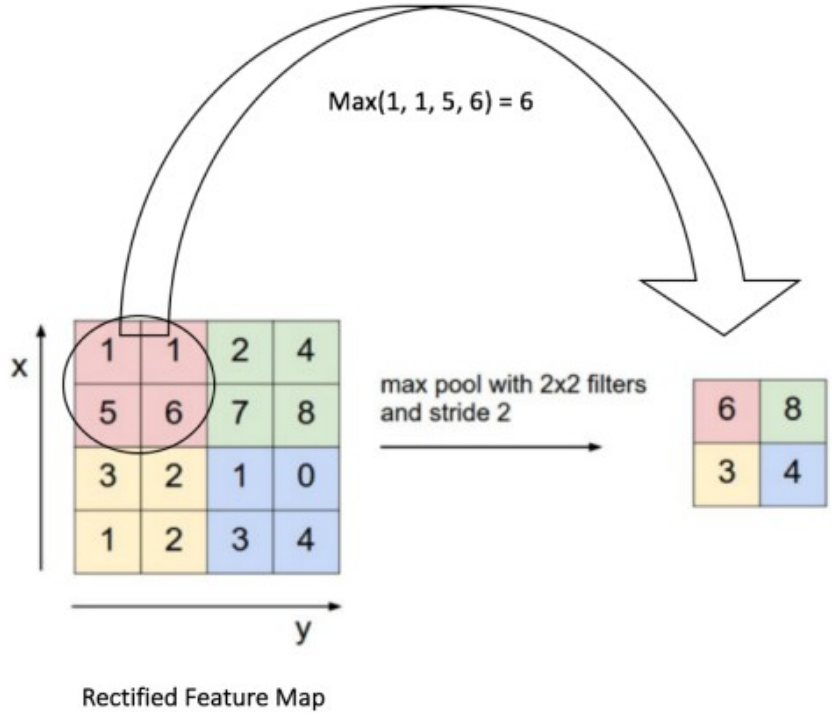


Abbildung 21: Max Pooling

Ein Neuronales Netzwerk besteht aber nur selten komplett aus Convolutional Layers. Üblicherweise stehen diese am Anfang des Netzwerks und gehen in den letzten Schichten in ein herkömmliches, vollständig vernetztes, mehrlagiges Perzeptron über, das komplexere Zusammenhänge zwischen den Features erkennen kann. (vgl. Karn, 2016)

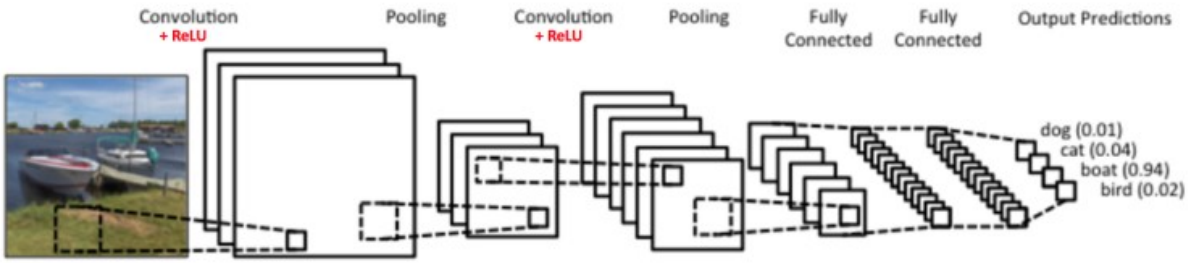


Abbildung 22: Struktur eines Convolutional Neural Network

2.5.4 Anwendungsmöglichkeiten

Da die Komplexität der möglichen Mustererkennung eines Neuronalen Netzes nur durch dessen Aufbau und daher hauptsächlich die Rechenleistung beschränkt ist, könnten auch die Anwendungsgebiete weiter nicht sein. Die häufigsten Anwendungen sind wohl Bilderkennung und Vorhersagen in komplexen Systemen, wie zum Beispiel in der Wirtschaft (vgl. Mahanta,

2017), aber durch Kombination mit anderen Lernverfahren oder bestimmten Netzstrukturen können sogar komplexe Spieleagenten, wie zum Beispiel bei Googles Projekt AlphaGo, oder Instanzen mit scheinbarer, graphischer oder musikalischer Kreativität entwickelt werden. (vgl. Hassabis, 2016; Landwehr, 2018; Johnson, 2015)

3 Unsupervised Learning

Oft will man einer KI auch Aufgaben stellen, für die es unmöglich oder nur mit großem Aufwand möglich ist, vorbereitete Trainingsdaten zu erstellen. In diesem Fall bieten sich Unsupervised Learning Verfahren an. Diese benötigen nämlich im Gegensatz zu Supervised Learning Verfahren nur unklassifizierte Trainingsdaten. Die Daten sollen selbständig sinnvoll in Klassen eingeteilt werden. (vgl. Ertel, 2016, S. 313ff.)

3.1 Clustering

Häufig bestehen in einer Datenmenge klar von einander getrennte Datenwolken bzw. Cluster. Ein typisches Beispiel wäre etwa eine Sammlung von Zeitungsartikeln. Diese können normalerweise recht eindeutig durch Faktoren wie zum Beispiel deren Semantik in verschiedene Themengebiete geteilt werden. Die händische Einteilung ist aber natürlich für den Menschen meist zu aufwändig. Daher verwendet man für solche Aufgaben üblicherweise Clustering-Algorithmen. Diese entscheiden welche Datenpunkte welchem Cluster, also welcher Klasse, angehören und, falls erwünscht, sogar wie viele Cluster in einer Datenmenge zu finden sind. (vgl. Ertel, 2016, S. 244f.)

3.1.1 k-Means

Beim k-Means-Verfahren ist die Clusteranzahl bereits bekannt. Wie schon durch den Namen ersichtlich werden k Cluster durch ihre Mittelwerte festgelegt. Dazu werden zuerst k Mittelpunkte üblicherweise zufällig initiiert. Dann werden alle Datenpunkte analog zum Nearest-Neighbour-Verfahren zum jeweils nächsten Cluster-Mittelpunkt klassifiziert und anschließend die Cluster-Mittelpunkte der neu entstandenen Cluster berechnet. Dies wird dann mit den jeweils neuen Mittelpunkten solange wiederholt bis keine Veränderung mehr auftritt.

k-Means$(x_1, ..., x_n, k)$
  initialisiere $\mu_1, ..., \mu_k$ (z.B. zufällig)
  Repeat
    Klassifiziere $x_1, ..., x_n$ zum jeweils nächsten μ_i
    Berechne $\mu_1, ..., \mu_k$ neu
  Until keine Änderung in $\mu_1, ..., \mu_k$
  Return$(\mu_1, ..., \mu_k)$

Die Cluster-Mittelpunkte werden dabei wie folgt berechnet:

$$\mu = \frac{1}{l} \sum_{i=1}^{l} x_i$$

(vgl. Ertel, 2016, S. 246f.)

Das k-Means hat damit natürlich den großen Nachteil stark von der zufälligen Mittelpunktinitiierung am Anfang abhängig zu sein. Um dies zu umgehen, kann der Algorithmus jedoch einfach mehrmals mit verschiedenen zufälligen Mittelpunkten ausgeführt werden und die beste Partition wird dann mittels mittels in Kapitel 3.1.3 beschriebenem Silhouette Width Kriterium ermittelt.

3.1.2 Hierarchisches Clustering

Beim hierarchischen Clustering wird zuerst jeder Punkt als ein eigener Cluster angenommen. Folgend werden nun laufend die jeweils nächsten Nachbarcluster miteinander vereint, solange bis eine Abbruchbedingung erreicht ist, etwa eine gewünschte Anzahl an Clustern oder ein gewünschter durchschnittlicher Cluster-Abstand.

Hierarchisches-Clustering$(x_1, ..., x_n)$
  initialisiere $C_1 = \{x_1\}, ..., C_n = \{x_n\}$
  Repeat
    Finde zwei Cluster C_i und C_j mit kleinstem Abstand
    Vereinige C_i und C_j
  Until Abbruchbedingung erreicht
  Return(resultierende Clustermenge)

Hierbei bieten sich natürlich wieder unterschiedliche Möglichkeiten an, den Abstand der Cluster zu messen. Üblicherweise werden die zwei nächsten Punkte der beiden Cluster verglichen, ebenso können aber auch die jeweils voneinander entferntesten Punkte und die Cluster-Mittelpunkte verwendet werden. Dies hat bei deutlich abgegrenzten Clustern zwar

kaum Auswirkung, kann jedoch bei undeutlichen Grenzen einen großen Unterschied machen. (vgl. Ertel, 2016, S. 248f.)

$$d_{\min}(C_i, C_j) = \min_{x \in C_i, y \in C_j} d(x, y)$$

$$d_{\max}(C_i, C_j) = \max_{x \in C_i, y \in C_j} d(x, y)$$

$$d_{\text{mean}}(C_i, C_j) = d(\frac{1}{m}\sum_{k=1}^{m} x_k, \frac{1}{n}\sum_{k=1}^{n} y_k) \quad ...x \in C_i, y \in C_j, m = |C_i|, n = |C_j|$$

3.1.3 Automatische Bestimmung der Anzahl der Cluster

Oft ist dem Anwender jedoch gar nicht bekannt, wie viele Cluster auf einer Datenmenge am sinnvollsten sind und er möchte einfach eine möglichst gute Partition, das heißt Cluster-Verteilung. Ein Algorithmus soll also entscheiden, wie viele Cluster in einer Datenmenge am sinnvollsten sind. Hierzu nimmt man an, dass in einer guten Partition der mittlere Abstand von Punkten innerhalb eines Clusters geringer sein sollte als der mittlere Abstand eines Punktes zu Punkten fremder Cluster. Dies kann durch das Silhouette Width Kriterium bemessen werden.

Um dessen Funktionsweise zu verstehen, wird zuerst die Funktion c definiert, die allen Datenpunkten $(x_1, ..., x_n)$ den jeweiligen Cluster zuordnet, also quasi die zu bewertende Partition:

$$c : x_i \mapsto c(x_i)$$

Ferner wird die Funktion

$$\overline{d}(x_i, c_\ell) = \frac{1}{n} \sum_{j=1, y_j \neq x_i}^{n} d(x_i, y_j) \qquad ...y \in c_\ell, n = |c_\ell|, c_\ell \in c$$

definiert, die den mittleren Abstand aller Punkte im Cluster c_ℓ zum Punkt x_i beschreibt.

Damit kann nun ein Maß für die Lage eines Punktes relativ zu seinem Cluster aufgestellt werden:

$$s(x_i) = \begin{cases} 0, & \text{falls} |c(x_i)| = 1 \\ \dfrac{\min\limits_{c_\ell \neq c(x_i)} \{\overline{d}(x_i, c_\ell)\} - \overline{d}(x_i, c(x_i))}{\max\{\min\limits_{c_\ell \neq c(x_i)} \{\overline{d}(x_i, c_\ell)\}, \overline{d}(x_i, c(x_i))\}}, & \text{sonst} \end{cases} \qquad ...c_\ell \in c$$

,

wobei $\min_{c_\ell \neq c(x_i)} \{\overline{d}(x_i, c_\ell)\}$ der mittlere Abstand des Punktes x_i zu allen Punkten des nächsten Clusters und $\overline{d}(x_i, c(x_i))$ der mittlere Abstand zu allen Punkten innerhalb des eigenen Clusters ist. Wenn also $s(x_i)=1$ gilt, liegt x_i in der Mitte seines eigenen Clusters. Bei $s(x_i)=0$, liegt x_i am Rand zwischen dem eigenen und einem fremden Cluster und bei $s(x_i)=-1$ liegt x_i völlig im „falschen“ Cluster.

Nach diesem Muster werden nun einfach alle Punkte bewertet und das arithmetische Mittel aus den Ergebnissen berechnet. Resultat ist das Silhouette Width Kriterium:

$$S = \frac{1}{n} \sum_{i=1}^{n} s(x_i)$$

Am effektivsten wäre nun natürlich alle möglichen Partitionen einer Datenmenge zu berechnen und diejenige mit dem maximalen Silhouette Width Kriterium auszuwählen. Dies ist jedoch nicht praktikabel, da die Zahl der Partitionen exponentiell mit der Zahl der Datenpunkte wächst. Stattdessen macht man sich heuristische Methoden zu Nutze. Möglich wäre zum Beispiel den k-Means Algorithmus für p Initialisierungen und jeweils k von 1 bis n durchlaufen zu lassen und nach Silhouette Width Kriterium die beste Partition auszuwählen. Dies ist im OMRk-Algorithmus realisiert:

```
OMRk(x_1, ..., x_n, p, k_max)
  S* = −∞
  For k = 2 To k_max
      For i = 1 To p
          Erzeuge eine zufällige Partition mit k Clustern (Initialisierung)
          Ermittle mit k-Means eine Partition P
          Bestimme S für P
          If S > S* Then
              S* = S; k* = k; P* = P
  Return(k*, P*)
```

(vgl. Ertel, 2016, S. 250f.)

3.1.4 Vorteile und Nachteile

Die obig vorgestellten Algorithmen sind alle für Lernverfahren ohne Lehrer (Unsupervised Learning) vergleichsweise einfach und effizient, dafür aber auch recht eingeschränkt in ihrer Arbeitsweise. Sie können eben nur klar von einander getrennte Datenwolken unterscheiden. Dagegen sind etwa AutoEncoder um einiges mächtiger, eine Form neuronaler Netze, die auf

Unsupervised Learning beruht und dabei Muster in Daten erkennen kann, die in ihrer Komplexität quasi nur durch die verfügbare Rechenleistung und die Netzstruktur beschränkt sind. (vgl. Chandradevan, 2017)

Für einfache Aufgaben ist herkömmliches Clustering jedoch völlig ausreichend und sowohl für den Programmierer, als auch bezüglich der Rechenzeit unaufwändiger und durch seine Einfachheit auch weniger fehleranfällig. Als Beispiele für solche Aufgaben sind hier etwa die thematische Sortierung von Webseiten durch Suchmaschinen oder die Fehleranalyse an einem Gerät anhand einfacher Sensoreninputs zu nennen. (vgl. Ertel, 2016, S. 244f.)

4 Reinforcement Learning

Wenn zu Beginn gar keine Trainingsdaten zur Verfügung stehen, aber der sogenannte Agent, also die KI, die Arbeitsumgebung erkunden kann, wird meistens eine Form des Reinforcement Learnings angewandt. Die Ausgabe der KI muss dabei einen Einfluss auf die Umgebung haben und ein Feedbacksystem muss beurteilen, ob die bewirkte Veränderung gut oder schlecht war. Durch Versuch und Irrtum soll der Algorithmus dann herausfinden, welche Aktion (bzw. welcher Output) sich in einer bestimmten Situation beziehungsweise in einem bestimmten State positiv oder negativ auswirkt. (vgl. Vorhies, 2016)

Ein Musterbeispiel für die Anwendung eines solchen Verfahrens ist das Erlernen von Computerspielen. Die meisten Computerspiele haben ein bereits eingebautes Feedbacksystem, zum Beispiel in Form eines Scores. Bei vielen Spielen erfolgt außerdem die Rückmeldung fast augenblicklich nach der Aktion, was das Lernverfahren meist stark vereinfacht. Außerdem sind die meisten Spiele deterministisch, das heißt eine Aktion zu einem bestimmten State führt immer zur selben Veränderung der Umgebung.

Die Umgebung wäre bei diesem Beispiel das Computerspiel, der State die momentane Bildschirmausgabe und die möglichen Aktionen das Betätigen der Controllertasten. (vgl. Simonini, 2016a)

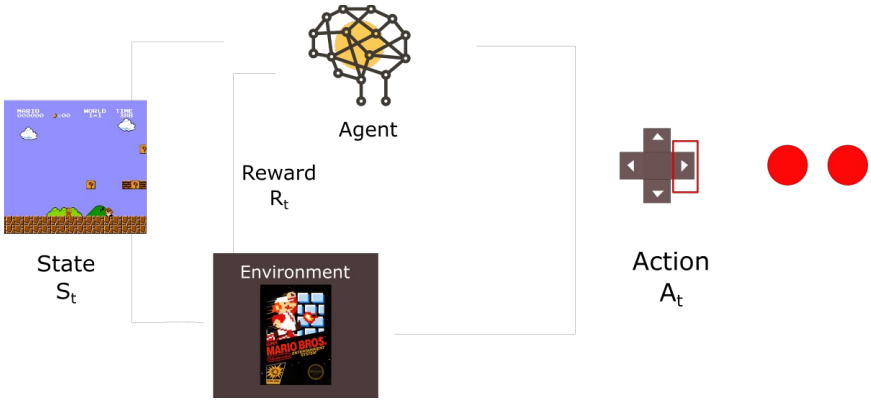


Abbildung 23: Die Komponenten des Reinforcement Learning am Beispiel Super Mario Bros.

Agent	Environment	Actions	Rewards	Policy
Board game player	Set of all possible game configs.	All legal moves	Win the game	Optimal strategy
Robot hand	Set of hand and finger positions	Bend/rotate wrist, close fingers, grasp tightly/loosely	Object is successfully picked up	Most efficient set of movements to pick up piece
Mouse	Maze	Running, turning	Get cheese	Most direct path to cheese
Credit card company	All customers in default	Set of collection actions	Cost of each attempt + reward for successful collection	Optimal strategy for debt collection
Marketing Team	All potential customers and ads that can be shown	Show an ad to a potential customer	Cost of placing an ad + value of successful sale	Optimal ad placement strategy
Room Temperature Control System	Sensor reading of room temperature	Adjust temp up, down, or not at all	Occupants are most satisfied with temp.	Optimal temp control solution.
Self-Driving Car	Sensor detects yellow light	All potential combination of accelerator and brakes	Avoids accident, remains within the law	Optimal strategy for stop/go at intersection
Robot Vacuum	Monitor on-board charge status	Continue vacuuming or return to base station	Does not get stranded without power	Best charging versus vacuuming performance
Call Center	Status of each customer in the queue	Connect customer to reps	Customer satisfaction	Optimal queueing strategy
Website Designer	Set of possible layout options	Changing the layout	Increased Click through rate	Ideal layout to optimize click throughs

Abbildung 24: Weitere Beispiele für Reinforcement Learning

Ziel des Agenten ist es, den Belohnungswert zu maximieren. Der erwartete Belohnungswert ist die Summe aller erwarteten Belohnungen in den folgenden States:

$$G_t = \sum_{k=0}^{T} R_{t+k+1}$$

Wobei spätere Belohnungen normalerweise weniger sicher sind. Daher werden diese durch den Faktor γ gewichtet:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \text{ , wobei } 0 \leqslant \gamma < 1$$

Es gibt drei Arten dieses Problem der Belohnungsmaximierung zu bewältigen: „value-based“, „policy-based“ und „model-based“.

Value-based Systeme haben eine Bewertungsfunktion, die jedem State oder jeder möglichen Aktion in einem State dessen beziehungsweise deren maximalen erwarteten Belohnungswert zuordnet. So kann für jeden State, in dem sich der Agent gerade befindet, der beste nächste State beziehungsweise die beste Aktion ausgewählt werden, indem immer das Maximum der Bewertungsfunktion gewählt wird.

Policy-based Systeme haben eine Policy Funktion, die direkt jedem State eine beste Aktion zuordnet. Eine Bewertungsfunktion ist meist relativ einfach in eine solche Policy Funktion umzuwandeln.

In model-based Systemen wird das Verhalten der Umgebung modelliert. Da sich jedoch jede Umgebung völlig verschieden verhält, gibt es hierfür keine allgemeinen Lösungsansätze oder Verfahren. (vgl. Simonini, 2016a)

4.1 Q-Learning

Q-Learning ist ein value-based Verfahren. Es wird dabei durch die Funktion $Q(s,a)$ jeder möglichen Aktion a zu einem State s ein Wert zugeordnet, der bestimmt, wie sinnvoll diese Aktion ist. Zu Beginn hat der Agent natürlich noch kein Wissen über die Umgebung, weshalb alle Aktionen noch mit 0 bewertet werden.

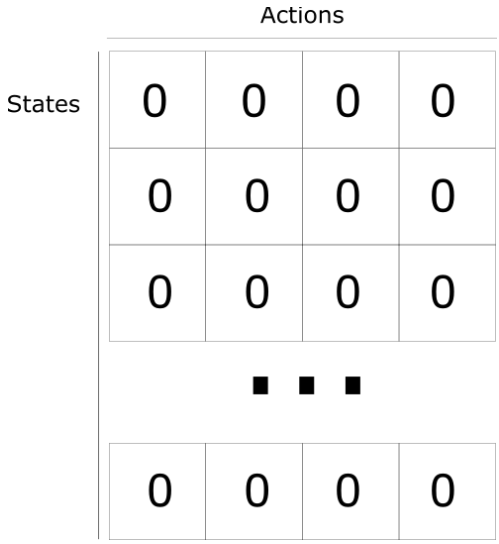


Abbildung 25: Eine neu initialisierte Q-Funktion in einer Tabelle dargestellt

Nach und nach werden Aktionen ausgeführt, die erhaltene Belohnung gemessen und der jeweilige Q-Wert der Aktion zu diesem State erneuert. Der neue Q-Wert wird nach der Bellman Formel berechnet:

$$NewQ(s,a)=Q(s,a)+\eta(R(s,a)+\gamma\, max\, Q'(s',a')-Q(s,a))$$

η ist eine Lernrate, $R(s,a)$ ist die erhaltene Belohnung, γ ist der Abfall des Wertes zukünftiger Belohnungen und $maxQ'(s',a')$ ist der maximale Q-Wert des neuen States.

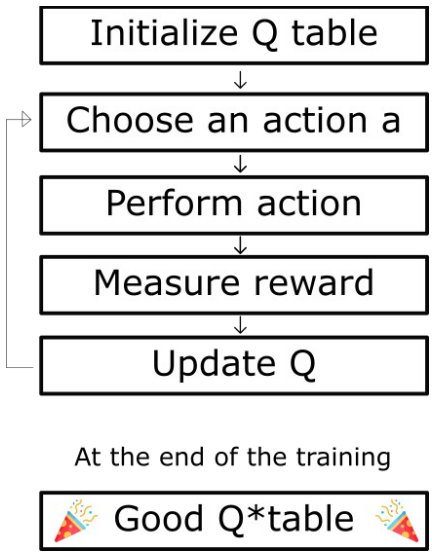


Abbildung 26: Der Q-Learning-Algorithmus

Da die Q-Funktion erst mit der Zeit sinnvolle Aussagen über die Umgebung treffen kann, werden zu Beginn alle Aktionen zufällig gewählt und erst mit der Zeit werden immer mehr Entscheidungen anhand des Maximums der Q-Funktion getroffen. (vgl. Simonini, 2016b)

4.2 Deep Q-Learning

In vielen Umgebungen gibt es jedoch nahezu unendlich viele States, was dazu führt, dass die herkömmliche Q-Funktion extrem lange braucht, bis sie sinnvolle Aussagen über die Umgebung treffen kann. Daher wird meistens diese tabellenartige Q-Funktion durch ein Neuronales Netz ersetzt, das den momentanen State und meistens auch einige vergangene States, um zeitabhängige Abläufe zu erkennen, aufnimmt und für alle möglichen Aktionen einen Q-Wert schätzt.

Das verwendete Netz ist meistens ein herkömmliches mehrlagiges Perzeptron oder, wenn der State durch Bilder beschrieben wird, ein Convolutional Neural Network. Der Sollwert der Endneuronen, den man benötigt, um die Fehlerfunktion und damit die Gradienten der Gewichte zu berechnen, ist der maximale Q-Wert des nächsten States durch γ gewichtet, also der maximale erwartete Belohnungswert.

Das Neuronale Netz wird genauso wie die normale Q-Funktion nach jeder ausgeführten Aktion angepasst. Auch beim Deep Q-Learning werden am Anfang alle, aber mit der Zeit immer weniger Aktionen zufällig gewählt, da auch das Neuronale Netz erst nach einiger Erkundung sinnvolle Aussagen über die Umgebung treffen kann. Wenn anhand des Netzes die Entscheidung getroffen werden soll, wird immer die höchst bewertete Aktion gewählt.

Da Deep Q-Learning sehr anfällig auf Overfitting ist, wird meistens außerdem ein Experience Replay Buffer angelegt, der manche bereits gelernten Erfahrungen speichert und aus dem später an zufällig gewählten Zeitpunkten wieder gelernt wird. (vgl. Simonini, 2016c)

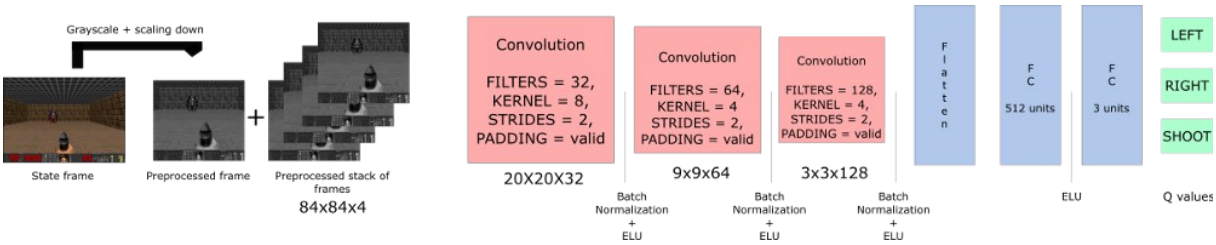


Abbildung 27: Struktur eines Neuronalen Netzes für Deep Q-Learning am Beispiel Doom

5 Fazit

Eine Instanz künstlicher Intelligenz nimmt Daten auf und versucht, diese nach einem gelernten Muster zu erkennen und daraus einen sinnvollen Output zu erschließen. Die dabei angewandten Lernverfahren kann man je nach notwendiger Beschaffenheit der Input-Daten in drei Kategorien einteilen:

Supervised Learning Verfahren benötigen bereits klassifizierte Trainingsdaten, also Datenpaare aus jeweils einerseits den Input-Daten des Beispiels und dem gewünschten Output. Aus vielen Beispielen versucht das Lernverfahren dann, allgemeine Regeln für das Verhältnis von Input und Output zu generalisieren. Im Kapitel 2 wurden hierfür als Beispiele das Nearest-Neighbour-Verfahren, Lernen von Entscheidungsbäumen, One-Class-Learning mit der Nearest-Neighbour-Data-Description, Perzeptronen und Neuronale Netze behandelt.

Das Nearest-Neighbour-Verfahren zeichnet sich vor allem durch seine Einfachheit aus. In der Lernphase speichert es alle Trainingsdaten direkt ab, was auch als Lazy Learning bezeichnet wird. In der Anwendungsphase sucht es dann aus dieser gelernten Datenmenge jeweils den nächsten Datenpunkt zum zu klassifizierenden Punkt und nimmt dessen Klasse als Output an.

Lernen von Entscheidungsbäumen hat im Vergleich zu anderen Lernverfahren den großen Vorteil, nach für den Menschen verständlichen Abläufen zu klassifizieren und macht damit nachträgliche Änderungen durch Menschen relativ einfach. Ein Entscheidungsbaum wird dazu üblicherweise von oben nach unten aufgebaut, wobei immer das Merkmal mit dem jeweils höchsten Informationsgewinn für den nächsten Knoten gewählt wird. Dazu muss der Informationsgehalt einer Datenmenge gemessen werden können, wozu das Merkmal der Entropie verwendet wird.

Falls nur für eine von zwei möglichen Klassen Trainingsdaten gesammelt werden können, muss man auf One-Class-Learning zurückgreifen. Die Nearest-Neighbour-Data-Description ist eine Abwandlung des Nearest-Neighbour-Verfahrens, die One-Class-Learning ermöglicht. Dieses Verfahren speichert ebenfalls alle Trainingsdaten direkt ab und vergleicht in der Anwendungsphase jeweils den Abstand des zu klassifizierenden Punkt und seinem nächsten Nachbarn in der Trainingsdatenmenge mit dem durchschnittlichen Abstand innerhalb der Trainingsdaten.

Perzeptronen sind zwar sehr eingeschränkt in der Komplexität der Muster, die sie erkennen können, sind jedoch trotzdem bis heute extrem wichtig im Bereich der Künstlichen Intelligenz, da sie die Basis für Neuronen in Neuronalen Netzen bilden, welche eines der meist genutzten Lernverfahren darstellen. Es multipliziert alle Koordinaten des Inputvektors mit einem jeweiligen Gewicht, addiert die Produkte auf und verarbeitet dieses Ergebnis schließlich mit einer Aktivierungsfunktion, meist einer einfachen Schwellenwertfunktion. Dieser Aufbau hat zur Folge, dass Perzeptronen nur mit zwei-klassigen, linear separierbaren Anwendungsproblemen umgehen können.

Wenn mehrere Perzeptronen jedoch zu komplexen Strukturen, sogenannten Neuronalen Netzen, verbunden werden, ist die Komplexität der zu erkennenden Muster in den Trainingsdaten nur mehr durch die Netzstruktur und die Rechenleistung des Computers beschränkt. Die einzige Abwandlung an Perzeptronen als Neuronen ist dabei die Verwendung einer Sigmoidfunktion als Aktivierungsfunktion, da diese differenzierbar ist, was für Backpropagation, den Lernalgorithmus Neuronaler Netze, unabdingbar ist. Bei der Backpropagation wird eine Fehlerfunktion aufgestellt, die den Fehler des Outputs im Vergleich zum Sollwert beschreibt. Dann wird von hinten durch das Neuronale Netz iteriert und für alle Gewichte ihre Auswirkung auf den Fehler gemessen. Mit diesen Erkenntnissen können dann alle Gewichte in die richtige Richtung verändert werden, was nach vielen Durchläufen an verschiedenen Beispielen zu einer richtigen Klassifikation führen sollte.

Unsupervised Learning Verfahren benötigen nur unklassifizierte Trainingsdaten. Diese teilen sie dann selbstständig in sinnvolle Klassen ein. Hierfür wurden in dieser Arbeit die Clustering-Verfahren k-Means, hierarchisches Clustering und automatische Bestimmung der Clusteranzahl durch das Silhouette Width Kriterium näher betrachtet. Clustering-Verfahren gehen von klar abgetrennten Datenwolken, sogenannten Clustern, in den Trainingsdaten aus.

Weiß man bereits die gewünschte Anzahl an Clustern, ist der wohl einfachste Algorithmus das k-Means-Verfahren. Dieses bestimmt die gewünschte Anzahl an Clustern zuerst durch zufällig gewählte Punkte und einer Nearest-Neighbour-Methode, berechnet die Mittelpunkte dieser Cluster und beginnt mit diesen Mittelpunkten wieder von vorne, solange bis sich nichts mehr verändert.

Beim hierarchischen Clustering muss ebenfalls die gewünschte Anzahl an Clustern oder ein anderes Abbruchkriterium, wie etwa der durchschnittliche Abstand der Cluster, bekannt sein.

Zuerst werden hierbei nämlich alle Punkte als ein eigener Cluster angenommen und dann jeweils die zueinander nächstgelegenen Cluster verbunden, solange bis das Abbruchkriterium erreicht ist.

Wenn die Anzahl der Cluster völlig automatisch bestimmt werden soll, werden üblicherweise durch andere Verfahren wie etwa dem k-Means-Verfahren oder hierarchischem Clustering verschiedene Partitionen, also Cluster-Einteilungen der Datenmenge, erstellt und durch das Silhouette Width Kriterium, das die Qualität einer Partition misst, die beste ausgesucht.

Reinforcement Learning ist die jüngste Art Künstlicher Intelligenz und bezeichnet Lernverfahren, die gar keine Trainingsdaten benötigen, sondern selbstständig durch Erkunden der Arbeitsumgebung und Trial-and-Error versuchen, einen vorgegebenen Belohnungswert zu maximieren. In den Kapiteln 4.1 und 4.2 wurde hier näher auf Q-Learning und Deep Q-Learning eingegangen.

Beim Q-Learning wird quasi ein große Tabelle erstellt, in der für jede mögliche Aktion zu jedem möglichen Zustand der Arbeitsumgebung die zu erwartende Belohnung für diese Aktion eingetragen wird. Diese Tabelle wird langsam aufgebaut, indem der Agent zuerst zufällig, später immer öfter durch die Tabelle entschiedene Aktionen ausführt und aus der eventuell dadurch erhaltenen Belohnung und der laut momentanen Tabelle zu erwartenden Belohnung im neuen Zustand der Arbeitsumgebung den Wert der ausgeführten Aktion ableitet und in der Tabelle einträgt.

Eine solche Tabelle ist natürlich bei vielen Anwendungen vor allem in einer realen Arbeitsumgebung, wo es quasi unendlich viele möglichen Zustände der Arbeitsumgebung gibt, nicht praktikabel. Weshalb diese oft nur durch ein neuronales Netz approximiert wird. Dies nennt man Deep Q-Learning. Ein solches Netz nimmt den Zustand als Input auf und gibt zu jeder möglichen Aktion den geschätzten Q-Wert aus.

Man erkennt schnell, wie weitverzweigt und komplex das Feld der Künstlichen Intelligenz ist. Daher konnte in dieser Arbeit natürlich auch keine vollständige Liste aller Lernmethoden Künstlicher Intelligenz behandelt werden. Es wurde jedoch die allgemein anerkannte Einteilung in Supervised, Unsupervised und Reinforcement Learning und jeweils wichtige Vertreter vorgestellt. In der Praxis werden durchaus auch Adaptionen und Kombinationen der vorgestellten Verfahren beziehungsweise maßgeschneiderte Verfahren angewendet.

Literaturverzeichnis:

Printmedien:

Ertel, Wolfgang: Grundkurs künstlicher Intelligenz: Eine praxisorientierte Einführung, 4. Auflage, Weingarten: Springer Vieweg, 2016.

Winston, Patrick Henry: Artificial Intelligence. Third Edition. Reading Mass.: Addison-Wesley, 1992.

Online zur Verfügung gestellte Quellen:

Champandard, Alex J.: Reinforcement Learning. o.J. http://reinforcementlearning.ai-depot.com/ [Zugriff: 03.01.2019]

Chandradevan, Ramraj: AutoEncoders are Essential in Deep Neural Nets. 2017. https://towardsdatascience.com/autoencoders-are-essential-in-deep-neural-nets-f0365b2d1d7c [Zugriff: 24.01.2019]

Copeland, B.Jack: artificial intelligence | Definition, Examples, and Applications | Britannica.com. 1998. https://www.britannica.com/technology/artificial-intelligence [Zugriff: 17.08.2018]

Grünschloß, Leonhard: Perzeptron. 2005. http://www.informatik.uni-ulm.de/ni/Lehre/WS04/ProSemNN/pdf/perzeptron.pdf [Zugriff: 29.12.2018]

Hassabis, Demis: AlphaGo: using machine learning to master the ancient game of Go. 2016. https://blog.google/technology/ai/alphago-machine-learning-game-go/ [Zugriff: 27.01.2019]

Johnson, Daniel: Composing Music With Recurrent Neural Networks. 2015. http://www.hexahedria.com/2015/08/03/composing-music-with-recurrent-neural-networks/ [Zugriff: 27.01.2019]

Karn, Ujjwal: An Intuitive Explanation of Convolutional Neural Networks. 2016. https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/ [Zugriff: 02.01.2019]

Karpathy, Andrej: Convolutional Neural Networks (CNNs / ConvNets). o.J. http://cs231n.github.io/convolutional-networks/ [Zugriff: 02.01.2019]

Landwehr, Tobias: Kreativität aus der Maschine. 2018. https://www.spektrum.de/news/kreativitaet-aus-der-maschine/1557286 [Zugriff: 27.01.2019]

Lee, Man Wai / Chrysostomou, Kyriacos / Chen, Sherry Y. / Liu, Xiaohui: Decision Tree Applications for Data Modelling (Artificial Intelligence). 2008. http://what-when-how.com/artificial-intelligence/decision-tree-applications-for-data-modelling-artificial-intelligence/ [Zugriff: 29.08.2018]

Loy, James: How to build your own Neural Network from scratch in Python. 2014. https://towardsdatascience.com/how-to-build-your-own-neural-network-from-scratch-in-python-68998a08e4f6 [Zugriff: 29.12.2018]

Mahanta, Jahnavi: Introduction to Neural Networks, Advantages and Applications. 2017. https://towardsdatascience.com/introduction-to-neural-networks-advantages-and-applications-96851bd1a207 [Zugriff: 27.01.2019]

Mazur, Matt: A Step by Step Backpropagation Example. 2015. https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/ [Zugriff: 30.12.2018]

Müller, Fabian: Das Rosenblatt Perzeptron – die frühen Anfänge des Deep Learnings. 2017. https://www.statworx.com/de/blog/das-rosenblatt-perzeptron-die-fruehen-anfaenge-des-deep-learnings/ [Zugriff: 17.08.2018]

Nielsen, Michael: Using neural nets to recognize handwritten digits. 2018a. http://neuralnetworksanddeeplearning.com/chap1.html [Zugriff: 29.12.2018]

Nielsen, Michael: How the backpropagation algorithm works. 2018b. http://neuralnetworksanddeeplearning.com/chap2.html [Zugriff: 29.12.2018]

Nowling, R. J.: Recommendation System Using K-Nearest Neighbors. 2016. https://rnowling.github.io/data/science/2016/10/29/knn-recsys.html [Zugriff: 27.01.2019]

Otte, Sebastian: Das Perzeptron. 2009. https://www.cs.hs-rm.de/~panitz/prog3WS08/perceptron.pdf [Zugriff: 29.12.2018]

Pantic, Maja: Course 395: Machine Learning – Lectures. 2011. https://ibug.doc.ic.ac.uk/media/uploads/documents/courses/ml-lecture4.pdf [Zugriff: 27.08.2018]

Quinlan, J. R.: Improved Use of Continuous Attributes in C4.5. 1996. https://arxiv.org/pdf/cs/9603103.pdf [Zugriff: 29.08.2018]

Ray, Michael: Nearest Neighbours: Pros and Cons. 2012. http://www2.cs.man.ac.uk/~raym8/comp37212/main/node264.html [Zugriff: 27.08.2018]

Sarkar, Manish / Leong, Tze-Yun: Application of K-nearest neighbors algorithm on breast cancer diagnosis problem. 2000. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2243774/pdf/procamiasymp00003-0794.pdf [Zugriff: 27.01.2019]

Shannon, Claude E.: A Mathematical Theory of Communication. 1948. http://math.harvard.edu/~ctm/home/text/others/shannon/entropy/entropy.pdf [Zugriff: 29.08.2018]

Shee, Hassan / Cheruiyot, Kipruto W. / Kimani, Stephen: Application of k-Nearest Neighbour Classification in Medical Data Mining. 2014. https://www.researchgate.net/publication/270163293_Application_of_k-Nearest_Neighbour_Classification_in_Medical_Data_Mining [Zugriff: 27.01.2019]

Simonini, Thomas: An introduction to Reinforcement Learning. 2018a. https://medium.freecodecamp.org/an-introduction-to-reinforcement-learning-4339519de419 [Zugriff: 03.01.2019]

Simonini, Thomas: Diving deeper into Reinforcement Learning with Q-Learning. 2018b. https://medium.freecodecamp.org/diving-deeper-into-reinforcement-learning-with-q-learning-c18d0db58efe [Zugriff: 03.01.2019]

Simonini, Thomas: An introduction to Deep Q-Learning: let's play Doom. 2018c. https://medium.freecodecamp.org/an-introduction-to-deep-q-learning-lets-play-doom-54d02d8017d8 [Zugriff: 03.01.2019]

Tong, Weida / Xie, Qian / Hong, Huixiao / Fang, Hong / Shi, Leming / Perkins, Roger / Petricoin, Emanuel F.: Using Decision Forest to Classify Prostate Cancer Samples on the Basis of SELDI-TOF MS Data: Assessing Chance Correlation and Prediction Confidence. 2004. http://europepmc.org/articles/PMC1247659 [Zugriff: 29.08.2018]

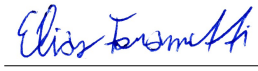
Vorhies, William: Reinforcement Learning and AI. 2016. https://www.datasciencecentral.com/profiles/blogs/reinforcement-learning-and-ai [Zugriff: 03.01.2019]

Abbildungsverzeichnis

Name: Elias Foramitti

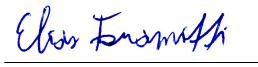
Selbstständigkeitserklärung

Ich erkläre, dass ich diese vorwissenschaftliche Arbeit eigenständig angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benutzt habe.

Blindenmarkt, 14.02.2019	
Ort, Datum	Unterschrift

Zustimmung zur Aufstellung in der Schulbibliothek

Ich gebe mein Einverständnis, dass ein Exemplar meiner vorwissenschaftlichen Arbeit in der Schulbibliothek meiner Schule aufgestellt wird.

Blindenmarkt, 14.02.2019	
Ort, Datum	Unterschrift

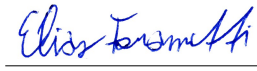
Begleitprotokoll

Name des Schülers: Elias Foramitti
Thema der Arbeit: Methoden künstlicher Intelligenz
Name der Betreuungsperson: Mag. Lukas Kerndler

Datum	Besprechungen mit der betreuenden Lehrperson, Fortschritte, offene Fragen, Probleme, nächste Schritte
20.06.17	Erstes in Kontakt Treten mit Herrn Prof. Kerndler bezüglich VWA-Betreuung und Vorschlag von Themenbereich KI
21.11.17	erste kurze Besprechung zu Thema und grober Gliederung
24.11.17	Vorbringung des Eingrenzungsvorschlag des Themas "Methoden künstlicher Intelligenz", der ausgearbeiteten groben Gliederung und Problemstellung; Unterschrift des schulinternen Bestätigungsformulars durch Herrn Prof. Kerndler
23.01.18	kurze allgemeine Besprechung zum Thema
30.01.18	Besprechung des offiziellen Einreichungsformulars
02.10.18	Besprechung der Kapitel Einleitung, Nearest Neighbour-Methode, Lernen von Entscheidungsbäumen und One-Class Learning
20.11.18	Besprechung des Kapitels Clustering und formaler Richtlinien
13.02.18	Finale Besprechung zu Formatierung, einigen inhaltlichen Aspekten und dem Abgabevorgang

Die Arbeit hat eine Länge von 52 312 Zeichen.

Blindenmarkt, 14.02.2019
Ort, Datum



Unterschrift des Schülers