Universität Bamberg
Angewandte Informatik

Seminar KI: gestern, heute, morgen

Die Evolution künstlicher neuronaler Netze

von

Johannes Rabold

24. Februar 2016

betreut von
Prof. Ute Schmid und Prof. Dietrich Wolter

Künstliche neuronale Netze sind ein Konzept der KI, entstanden in den 40ern, das es sich zur Aufgabe gemacht hat, typische Probleme der künstlichen Intelligenz mit einem Ansatz zu lösen, der auf dem biologischen Vorbild der neuronalen Struktur im Gehirn basiert. Wir werden die Anfänge mit den McCulloch-Pitts-Neuronen anschneiden, dann die Klassifikatoren single-layer- und multi-layer-Perzeptrons genauer betrachten und analysieren und zuletzt noch einen Blick auf heutige Bestrebungen werfen, besonders im Gebiet des deep learning.

1 Einleitung

Das Teilgebiet der künstlichen neuronalen Netze (im Folgenden KNNs genannt) beschreibt ein Konzept des Machine Learnings, das schon bis in die Anfangsphase erster elektronischer Rechner zurück reicht. Es spiegelt den Traum vieler Forscher des Fachbereiches der KI wieder, ein beobachtendes, schlussfolgerndes und vor allem lernendes rudimentäres Gehirn zu modellieren. Dieser Enthusiasmus wurde zwar im Laufe der Zeit oftmals auf eine harte Probe gestellt; der eigentliche Traum hat aber bis heute nicht an seinem Zauber verloren, was die vielfältige Verwendung des Prinzips in heutigen Anwendungsgebieten zeigt.

Traditionell wollte man erreichen, die universell einsetzbaren logischen Funktionen als ein Netz aus elementaren logischen Bausteinen zusammenzusetzen. Heutige KNNs beschäftigen sich vor allem mit einem typischen Problem der künstlichen Intelligenz: der Klassifikation von Daten. Diese Daten können vielfältiger Natur sein. Eine typische Anwendung von KNNs besteht aber in der Erkennung von handschriftlichen Zeichen oder natürlicher Sprache. Auch das Problem der Objekterkennung auf Bildern hat einen weiteren Teilbereich mit neuartigen Konzepten hervorgebracht.

In dieser Seminararbeit soll der Frage nachgegangen werden, welche grundlegenden Prinzipien (sowohl mathematisch als auch biologisch) hinter KNNs stehen, welchen Wandel und welche Neuerungen sich auf jeder Stufe der „Evolutionsleiter“ ergeben haben und zuletzt, wie sich KNNs in der Zukunft wohl weiter entwickeln werden.

2 Biologischer Hintergrund

Will man die einzelnen Designentscheidungen früherer sowie fortgeschrittenerer KNNs in ihrer Gesamtheit verstehen, muss man sich natürlich zunächst einmal das biologische Vorbild ansehen: die neuronalen Strukturen im Gehirn. Im nachfolgenden Abschnitt soll ein Blick auf die atomaren Schaltelemente, die Neuronen, geworfen werden und auf die Verknüpfung einzelner Neuronen zu einem schlussfolgerndem Netzwerk eingegangen werden. Dieser Einblick ist keinesfalls allumfassend und wurde an vielen Stellen nur sehr oberflächlich gehalten, damit der Fokus auf die wesentlichen Details, die für das Verständnis von KNNs wichtig ist, erhalten bleibt.

Ein einzelnes Neuron besteht im grundlegenden Aufbau aus einem Zellkörper, dem sogenannten Soma, vielen kleinen Verästlungen, die aus dem Soma hervorgehen und Dendriten genannt werden, und einem langen Fortsatz, dem Axon. Das Axon dient der Reizweiterleitung von elektrischen Signalen zu anderen Neuronen oder zu den Muskeln und kann teilweise Längen von über einem Meter erreichen. Am Ende des Axons befinden sich viele kleine Knöpfchen, die man präsynaptische Endigungen nennt und die die Schnittstelle zu anderen Neuronen bilden. Diese Endigungen docken quasi an die Dendriten von vielen anderen Neuronen an und verteilen so ihren elektrischen Impuls an die nachgeschalteten Neuronen[1] [(springer.com, oJ)].

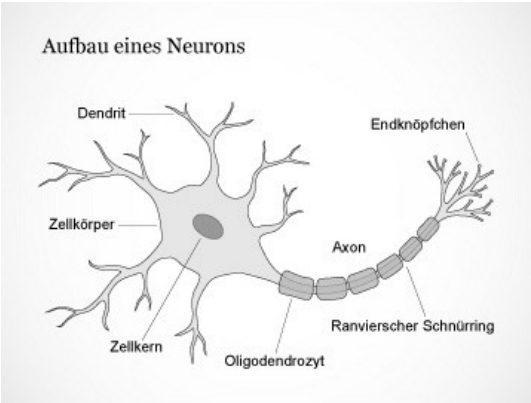


Abbildung 1: Der grundlegende Aufbau eines biologischen Neurons

Damit ein Neuron überhaupt einen elektrischen Impuls am Axon entlang schicken kann, muss die Summe aller an den Dendriten ankommender elektrischer Impulse einen gewissen Schwellenwert überschreiten. Hierbei muss man noch beachten, dass es sowohl exzitatorische (also reizverstärkende) so wie inhibitorische (reizhemmende) Neuronen gibt. In die Summe fließen Reizimpulse von inhibitorischen Neuronen negativ ein.

[1] In der biologischen Wirklichkeit geschieht diese Übertragung in den meisten Fällen auf chemischem Weg mit Hilfe von Neurotransmittern, die durch eine Lücke zwischen Präsynapsen und Dendriten, dem sogenannten synaptischen Spalt, geschickt werden.

Wird der Schwellenwert letztendlich erreicht, schickt das Neuron den vollen Impuls durch das Axon in Richtung der präsynaptischen Endigungen (auch Endknöpfchen genannt). Erfolgt kein Überschreiten des Schwellenwertes, so feuert das Neuron überhaupt nicht („Alles oder nichts-Prinzip“) [(springer.com, oJ)].

3 Übertragung auf die mathematisch-konzeptuelle Ebene

Will man das vorangegangene biologische Prinzip auf ein maschinell verarbeitbares mathematisches Modell übertragen, muss man erst ein paar Vereinfachungen vornehmen. Diese sollen ferner auch dazu dienen, KNNs auf die oben genannten Anwendungsgebiete zuzuschneiden. Dies bedeutet, man muss eine vernetzte Struktur modellieren, die einerseits eine n-elementige Eingabe erwartet, andererseits eine m-elementige Ausgabe produziert. Dazwischen muss irgendeine Art von Vernetzung stattfinden. Die elementarste Art der „Vernetzung“ besteht darin, n Eingaben mit m = 1 Ausgaben einfach zu verbinden. Bezogen auf die biologische Analogie bedeutet dies, dass wir genau ein Neuron vor uns haben, welches n Dendriten besitzt (also n Reize empfangen kann) und daraus 1 resultierenden Reiz berechnen kann.
Hier kommt dem Schwellenwert aus dem biologischen Vorbild eine wichtige Bedeutung zu. Man sagt, ein künstliches Neuron „feuert“ genau dann, wenn die Summe aller Eingangsreize x_i den Schwellenwert t des Neurons erreicht oder übersteigt, also

$$\sum_{i=1}^{n} x_i \geq t$$

[(uni muenster.de, oJ)]

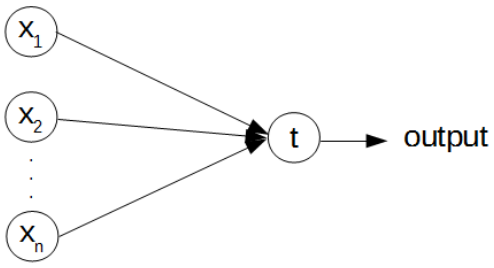


Abbildung 2: Ein einzelnes Neuron mit Eingangsreizen x_i und Schwellenwert t.

Man kann nun beliebig viele Neuronen hintereinander schalten, indem man die Ausgabe eines Neurons als Eingangsreiz eines anderen sieht. So bildet sich nach und nach ein Netz aus, welches imstande ist, ein Eingangsproblem in eine Ausgabe zu transformieren. Die Planung der Architektur dieses Netzes (damit ist hier die Verschaltung der einzelnen Neuronen sowie die jeweiligen Schwellenwerte gemeint) ist hierbei entscheidend für die Effektivität des Netzes im Bezug auf seine Problemlösefähigkeit. Wir werden im Verlauf

dieser Arbeit sehen, dass es verschiedene Arten von Verschaltungen von Neuronen sowie auch verschiedene Möglichkeiten gibt, einen resultierenden Reiz zu berechnen.

4 Historie

4.1 McCulloch & Pitts-Neuronen

Eine der ersten Pioniere auf dem Gebiet der künstlichen neuronalen Netze waren der Neurophysiologe und Kybernetiker Warren McCulloch und der Logiker und Psychologe Walter Pitts. In ihrem 1943 erschienenen Artikel „*A logical calculus of the ideas immanent in nervous activity* “ schufen sie zum ersten Mal eine mathematisch-logische Grundlage zur Erklärung von neuronalen Aktivitäten innerhalb eines kleinen Ausschnittes des Neuronennetzes.

Die Annahmen von McCulloch und Pitts stützen sich darauf, dass das Feuern eines Neurons ein Älles-oder-nichts-Prozessïst. Nur wenn genug exzitatorische Reize an den Dendriten anliegen, feuert das Neuron. Sie postulierten weiterhin, dass der Schwellenwert, den diese Annahme bedingt, eine intrinsische Eigenschaft jedes Neurons ist und dieser außerdem nicht von früherer Aktivität oder der Position des Neurons abhängt. Diese Überlegungen führten sie zur Beschreibung des mathematischen Modells der *McCulloch-Pitts-Neuronen* [(McCulloch and Pitts, 1943)].
McCulloch und Pitts nahmen an, dass jede logische Funktion, die bestimmte Eigenschaften erfüllt, von einem Netzwerk aus solchen Neuronen berechnet werden kann. Als Beispiel, um die Funktionsweise Dieser zu veranschaulichen, dient uns die logische AND-Funktion, bei der die Eingabe aus n = 2 logischen Wahrheitswerten (true, false) besteht und die Ausgabe das Ergebnis der AND-Funktion für die Eingabewerte darstellt (m = 1).

Tabelle 1: Die logische AND-Funktion

x_1	x_2	Ausgabe
false	false	false
false	true	false
true	false	false
true	true	true

Diese Funktion kann man als einzelnes Neuron modellieren, wobei man sich die Eingabewerte als Reize an den Dendriten vorstellt (Reiz ist vorhanden = *true/1*; Reiz bleibt aus = *false/0*) und den Ausgabewert als Ergebnis der Reizweiterleitung an der Axon-Endigung begreift.
Für die Berechnung einer korrekten Ausgabe addiert man zunächst die Inputwerte auf, wobei ein *true* für eine '*1*' und ein *false* für eine '*0*' steht. Jetzt kommt der Schwellenwert zum Tragen, den man für die AND-Funktion auf t = 2 setzt. Dieser Wert macht dann

Sinn, wenn man sich die Wahrheitstabelle (Tabelle 1) ansieht. Für alle Eingabepaare außer der letzten Zeile ergeben sich Summen unter 2 und dementsprechend als Ausgabe ein *false* (Das Neuron erreicht nicht seinen Schwellenwert und feuert nicht). Mit der letzten Zeile erreicht man den Schwellenwert 2 exakt und das Neuron feuert (Es produziert eine *1* als Ausgabe).

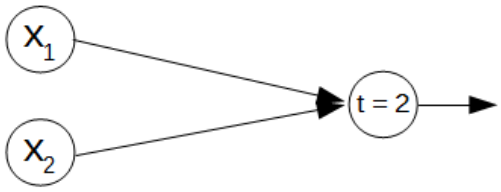


Abbildung 3: Die AND-Funktion modelliert als McCulloch-Pitts-Neuron mit festem Schwellenwert 2.

Es lässt sich leicht zeigen, dass für andere logische Grundoperationen ein ähnliches Prinzip angewendet werden kann (z.B. OR-Funktion: t = 1). Allgemein kann man nun zur Modellierung komplexer logischer Funktionen einzelne Neuronen mit ihren entsprechenden Schwellenwerten hintereinander schalten und die jeweilige einstellige Ausgabe als einzelne Eingabe für ein nachfolgendes Neuron verwenden.

Für die NOT-Funktion muss man allerdings noch inhibitorisch wirkende Neuronen einführen, die bei Aktivierung ein nachfolgendes Neuron komplett am Feuern hindern, selbst wenn alle anderen Eingabewerte summiert den Schwellenwert erreichen oder übersteigen.

Allgemein kann man sagen, dass McCulloch-Pitts-Neuronen, da sie nach dem „Alles-oder-nichts-Prinzip“ arbeiten, entweder voll aktiviert sind oder überhaupt nicht feuern. Die *Aktivitätsfunktion*, die den Ausgabereiz als eine Funktion der Eingabereize modelliert, bildet hier also eine Treppenfunktion. Alternativ kann man aber natürlich jede beliebige Funktion als Aktivitätsfunktion einsetzen. Denkbar sind zum Beispiel die Identitätsfunktion oder die logistische Funktion. So verlässt man aber nun den Problemraum der dichotomen Logik und betritt den reellen Zahlenraum.

McCulloch-Pitts-Neuronen waren also ein erster Versuch, die Aktivität biologischer Neuronen in einen mathematischen Rahmen zu setzen und so für die künstliche Intelligenz nutzbar zu machen. Gleichzeitig liefern diese Erkenntnisse ein Modell für die biologische Psychologie, das beschreiben kann, wie eine neuronale Struktur prinzipiell in der Lage sein kann, logische Schlussfolgerungen zu ziehen. Eine wichtige Funktion jedes neuronalen Netzwerkes, nämlich die Fähigkeit der Selbstmodifikation („Lernen“) ist aber noch nicht implementiert.

4.2 Hebb'sche Lernregel

„Lernen ist der Prozeß, der zu einer relativ stabilen Veränderung von Reiz-Reaktions-Beziehungen führt“. Ein Gehirn lernt, indem es durch Beobachten der Umwelt seine innere Struktur ändert, mit dem Ziel ein erwünschtes Verhalten an den Tag zu legen [(uni due.de, oJ)].

Wie so eine Veränderung der inneren Struktur aussehen könnte, versuchte zum ersten Mal der Psychologe Donald Hebb im Jahre 1949 zu erklären. Textuell lautet der Algorithmus, den er vorschlug etwa:

"Wenn ein Axon der Zelle A nahe genug ist, um eine Zelle B zu erregen und wiederholt oder dauerhaft sich am Feuern beteiligt, geschieht ein Wachstumsprozeß oder metabolische Änderung in einer oder beiden Zellen dergestalt, daß A's Effizienz als eine der auf B [...] feuernden Zellen anwächst."[(uni muenster.de, oJ)]

Wenn also zwei miteinander verbundene Neuronen zur selben Zeit stark feuern, sollte sich also die Verbindung zwischen den beiden Neuronen auch in gleichem Maße verstärken. Für einen Reiz, der von Neuron A auf die nachgeschalteten Neuronen B („Partner“ im Lernprozess) und C verteilt wird, bedeutet dies, dass nach dem Lernprozess die Präferenz, den Reiz auf A abzugeben gestiegen ist. Im Kontext von KNNs äußert sich eine solche Präferenz in der Einführung einer Gewichtung zwischen zwei Neuronen, die eine synaptische Verbindung besitzen. Ein Lernprozess findet nun statt, indem man diese Gewichtung verändert und zwar abhängig davon, wie stark der Ausgabereiz des vorgeschalteten Neurons ist, und wie stark das nachgeschaltete Neuron gerade aktiviert ist.

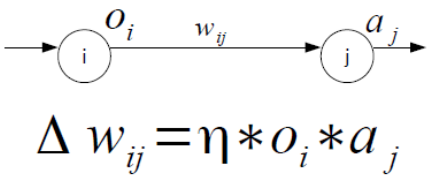


Abbildung 4: Die hebb'sche Lernregel als Änderung der Gewichtung zwischen zwei künstlichen Neuronen.

Mathematisch gesehen lautet die Änderung in der Gewichtung also:

$$\Delta w_{ij} = \eta * o_i * a_j$$

Hier ist o_i der Output-Reiz des vorangeschalteten Neurons, a_j die derzeitige Aktivierung des nachgeschalteten Neurons und η die sogenannte Lernrate, ein konstanter Wert, der die Lerngeschwindigkeit angibt.
Problematisch bei dieser Lernregel ist die Tatsache, dass die Gewichtung theoretisch ins Unendliche steigen kann. Kleinere Korrekturen können dies aber verhindern. Die hebb'sche Lernregel ist bis heute Grundlage für viele moderne Lernalgorithmen in künstlichen neuronalen Netzen [(uni muenster.de, oJ)].

4.3 Perzeptrons

4.3.1 Single-Layer-Perceptrons (Rosenblatt)

Anfang der Sechziger Jahre entwickelte der amerikanische Psychologe und Informatiker Frank Rosenblatt seinen sogenannten Perzeptron-Algorithmus zum Lernen eines binären Klassifikators und nutzte dabei die hebb'sche Lernregel. Ein Perzeptron ist in seiner einfachen (*single-layer-*) Form ein künstliches neuronales Netz zur Klassifizierung von Daten, welches aus einer Schicht von Eingangsneuronen besteht, an die man die einzelnen Instanzen des zu klassifizierenden Datensatzes anlegt, und einem Klassifikator-Neuron, welches gewichtete Verbindungen zu den Eingangsneuronen besitzt. Das Klassifikator-Neuron entscheidet mit einer Schwellenwerts-Funktion, zu welcher Klasse die Instanz der Trainingsdaten gehört. Hierbei werden zunächst die Eingangsreize mit der jeweiligen Gewichtung multipliziert und dann die Ergebnisse aufsummiert. Dann wird bestimmt, ob die Summe den Schwellenwert t erreicht oder übersteigt, oder nicht (binäre Klassifikation) [(uni muenster.de, oJ)].
Es ist meist komfortabler, die verwendete Treppenfunktion dergestalt zu transformieren, dass der „Schwellenwert" bei 0 liegt. Zu diesem Zweck berechnet man den sogenannten *bias* $b = -t$, und lässt diesen in die Summe einfließen. Hierbei tut man so, als existiere ein Eingangsreiz x_0 mit konstantem Wert 1 und einer Gewichtung mit dem Wert $w_0 = b$ [(uni freiburg.de, oJ)].

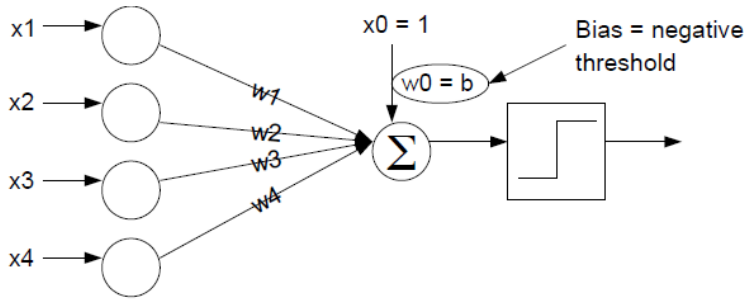


Abbildung 5: Ein Single-Layer-Perzeptron zur binären Klassifizierung von Daten.

Der Perzeptron-Algorithmus von Rosenblatt lernt auf Basis von vielen Trainingsdaten seine Parameter *Gewichtungen* und *Schwellenwert*. Bei jeder Instanz aus den Daten ist auch die gewünschte Klasse mit angegeben. Perzeptrons versuchen den Fehler, den sie zur Zeit noch mit ihrer Ausgabe produzieren, mit jeder weiteren Trainings-Instanz zu minimieren und so die Ausgabe immer näher an die gewünschte Klasse hin zu verschieben („*supervised learning, Lernen mit Lehrer*"). Ziel ist es schlussendlich, dem Perzeptron einen neuen, bisher unbekannten Datensatz zu präsentieren, den das Perzeptron dann richtig klassifiziert. Als Beispiel soll hier folgender fiktiver Datensatz dienen, bei denen 4 Personen ein Betriebssystem nach den Kriterien $x_1 = Usability$ und $x_2 = LevelofCustomisation$ bewertet hatten. Die erwünschte Klasse gibt an, um welches von zwei Betriebssystemen es sich jeweils gehandelt hatte.

Tabelle 2: Ein Trainingsdatensatz zum Lernen eines Perzeptrons

x_1	x_2	Erwünscht
0.8	4.5	0
2.4	3.3	0
2.9	2.0	1
4.1	0.9	1

Der Algorithmus läuft nun folgendermaßen ab:

- Initialisiere alle Gewichte (auch w_0) mit 0. Setze die Lernrate $\eta = const.$ (hier $\eta = 0.2$)
- Wiederhole solange, bis alle Trainingsdaten korrekt klassifiziert wurden:
  - Berechne den Output des Perzeptrons $o_t = sig(\sum_{i=0}^{2} w_i * x_i)$
  - Für alle Attribute berechne:
    - $w_i = w_i + \eta(d_t - o_t) * x_i$

Hierbei sind o_t die Ausgabe des Perzeptrons für den jeweiligen Trainingssatz, sig die Signum-Funktion, w die Gewichte, x die Merkmalsausprägungen und d_t die gewünschte Klasse für den Trainingssatz.

Das Herzsstück dieses Lernalgorithmus bildet die Anpassung der Gewichte w_i. Der alte Wert wird hier so geändert, dass das Ergebnis des Perzeptrons in Richtung der gewünschten Klasse wandert. Hierzu wird die Differenz des Ist-Ergebnisses zum Soll-Ergebnis berechnet und zur derzeitigen Gewichtung addiert. Mit der Lernrate kann hier die Geschwindigkeit einstellen, mit der das Perzeptron seinen Idealzustand erreicht. Man sollte jedoch keine zu großen Werte einstellen, da es sonst passieren kann, dass mit jeder Iteration der Wert über das Ziel "hinausschießt". Andererseits können zu kleine Werte dazu führen, dass das Perzeptron zu lange braucht, um zu einem Idealzustand zu gelangen [(uni freiburg.de, oJ)].

Wenn man den Algorithmus nun beispielsweise auf den dritten Datensatz anwenden würde, würde man zunächst $o_t = sig(0 * 1 + 0 * 2.9 + 0 * 2.0) = 0$ berechnen. Dieser Datensatz wurde also falsch klassifiziert. Das bedeutet, man muss die Gewichte anpassen:

- $w_0 = 0 + 0.2 * (1 - 0) * 1 = 0.2$
- $w_1 = 0 + 0.2 * (1 - 0) * 2.9 = 0.58$
- $w_2 = 0 + 0.2 * (1 - 0) * 2.0 = 0.4$

Der Algorithmus läuft nun solange, bis alle Trainingsdaten korrekt klassifiziert wurden. Am Schluss haben wir folgende Gewichte gelernt:

- $w_0 = 0$
- $w_1 = 0.42$
- $w_2 = -0.5$

Dies entspricht der folgenden Output-Gleichung für das Perzeptron:

$$o = 0.42 * x_1 - 0.5 * x_2$$

oder umgestellt

$$x_2 = 0.84 * x_1$$

Die geometrische Interpretation des gelernten Perzeptrons entspricht also folgender Geraden:

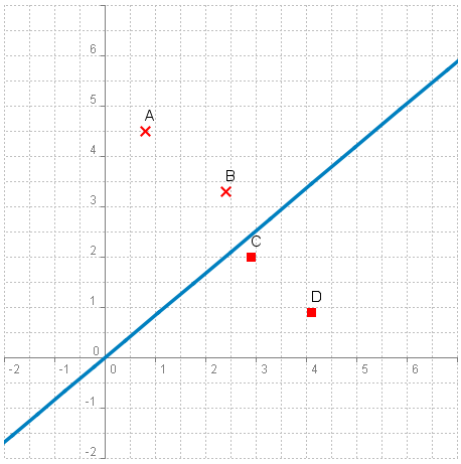
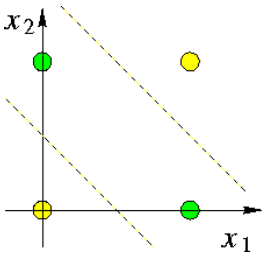


Abbildung 6: Geometrische Interpretation eines Perzeptrons mit zwei Input-Neuronen. Die Punkte stellen einige Testdaten dar.

Mit im Bild sind die Trainingsdaten. Man erkennt, dass die Perzeptron-Gerade die beiden Klassen voneinander trennt. Wenn nun ein neuer, bisher unbekannter Wert durch das Perzeptron klassifiziert wird, ist die Wahrscheinlichkeit sehr groß, dass es den Wert korrekt klassifiziert.

Für das obige Problem mit zwei Attributen (und dementsprechend zwei Input-Neuronen) kann das Perzeptron durch eine Gerade repräsentiert werden. Für höherdimensionierte Probleme (3-n Klassifikations-Attribute) ändert sich die Repräsentation entsprechend in eine Ebene oder n-dimensionale Hyperebene als Trennelement.
Damit der Perzeptron-Lernalgorithmus terminiert, muss das ihm gestellte Problem *linear separierbar* sein. Das bedeutet, die einzelnen Klassen jedes möglichen Testdatensatzes müssen in der geometrischen Interpretation durch eine Gerade, Ebene, Hyperebene oder

sonstige plausible geometrische Struktur trennbar sein. Ist dies nicht der Fall, terminiert der Algorithmus nicht. Ein einfaches Beispiel für ein Problem, welches nicht linear separierbar ist, ist das Lernen der logischen XOR-Funktion. Die Datenpunkte der Funktion lassen sich nicht mit einer Geraden in die entsprechenden Klassen *true* und *false* aufteilen und sind so nicht von einem single-layer-Perzeptron lernbar [(uni freiburg.de, oJ)].



x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

Abbildung 7: Die XOR-Funktion.

Dieses sogenannte *XOR-Problem* wurde von Marvin Minsky und Seymour Papert zum ersten Mal aufgezeigt und führte zum „Todesstoß“ der KNNs. Dies bedeutete, dass das gesamte Gebiet der KNNs für Jahre zum Erliegen kam [(uni muenster.de, oJ)].

4.3.2 Multi-Layer-Perceptrons

Das Forschungsfeld erlebte aber 1986 ein Comeback mit einem revolutionären Algorithmus, der imstande war, ein Perzeptron zu lernen, welches nicht nur aus einer, sondern mehreren Ebenen besteht (*multi-layer-perceptrons*). Solche Perzeptrons sind in der Lage, jedes Klassifizierungsproblem zu lösen (sowohl binär als auch mit mehreren Klassen).
Die allgemeine Architektur der multi-layer-Perzeptrons besteht aus einer Input-Schicht, einer Output-Schicht und 1-n Hidden-Schichten, wobei konsekutive Schichten voll miteinander vermascht sind. Jede Schicht außer der Output-Schicht enthält außerdem noch ein Bias-Neuron. Die Verbindungen sind gewichtet. Wie schon bei single-layer-Perzeptrons lernt die Struktur, indem sie Gewichte anpasst [(uni muenster.de, oJ)].

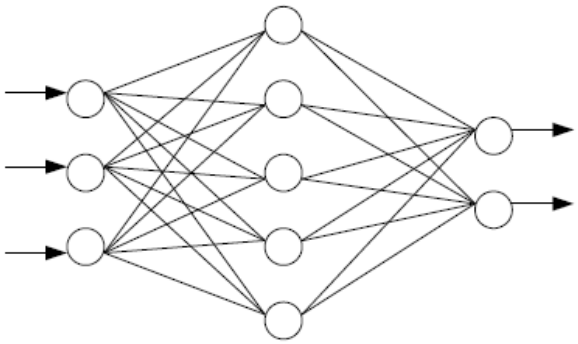


Abbildung 8: Ein multi-layer-Perzeptron

In Abbildung 8 ist ein multi-layer-Perzeptron abgebildet, welches einen Datensatz klassifizieren kann, der 3 Attribute besitzt und auf 2 Klassen aufgeteilt werden kann. Außerdem besitzt das Perzeptron eine Hidden-Schicht mit 5 Neuronen, welche imstande sind, gewisse Features eines Datensatzes zu repräsentieren. Wie dies genau vonstatten geht, ist hier zweitrangig, wichtig ist nur, dass je mehr Hidden-Schichten es gibt und je mehr Neuronen darin enthalten sind, die Klassifizierung besser funktioniert. Die Bias-Neuronen wurden hier weggelassen.

Allgemein funktioniert die Weiterleitung eines Reizes hier ähnlich wie in einem single-layer-Perzeptron. Zunächst werden alle ankommenden Reize von der vorgeschalteten Schicht addiert, dann wird die Aktivitätsfunktion auf die Summe angewendet und schlussendlich das Ergebnis an alle Neuronen der nachfolgenden Schicht propagiert. Die Aktivitätsfuntion muss hier jedoch differenzierbar sein (das ist wichtig für den später vorgestellten Lernalgorithmus). In Anlehnung an die Treppenfunktion nimmt man hier meist die logistische Funktion (siehe Abbildung 10) [(Nielsen, 2016)].

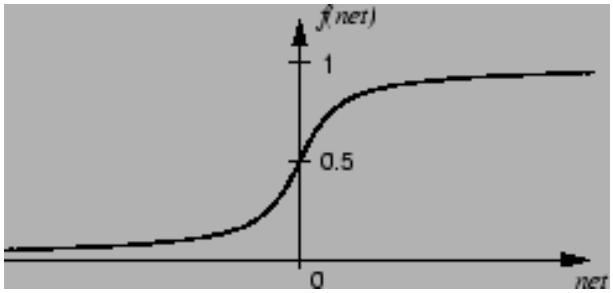


Abbildung 9: Die logistische oder sigmoide Funktion.

Das Lernverfahren für multi-layer-Perzeptrons ist der *Backpropagation-Algorithmus*. Er wurde 1974 von Paul Werbos erstmals vorgestellt, fand aber erst breiten Anklang mit den Veröffentlichungen von Rumelhart und McClelland 1986. Das Verfahren basiert darauf, einen perzeptroneigenen Fehlerterm (manchmal auch Kostenfunktion) zu minimieren, der die Diskrepanz zwischen erwünschter und derzeitiger Ausgabe darstellt. Dieses Bestreben setzt den Algorithmus wieder in die Kategorie des supervised learning.
Allgemein läuft der Backpropagation-Algorithmus wie folgt ab:

- Wiederhole für alle Trainings-Beispiele:
  - Propagiere die Eingabewerte durch das Netzwerk
  - Vergleich die Ausgabe-Werte mit den erwünschten Werten
  - Berechne den Fehlerterm
  - Propagiere den Fehler zurück in das Netzwerk
  - Passe die Werte proportional zu dem Einfluss an, den sie auf den Fehlerterm haben.

[(vgl. Chauvin and Rumelhart, 1995)]

Der Fehlerterm ist meistens die Summe der mean-squared-errors von erwünschter und eigentlicher Ausgabe über alle Output-Neuronen. Zur Berechnung des Einflusses eines Gewichts auf den Fehlerterm arbeitet man mit den partiellen Ableitungen des Fehlerterms nach der entsprechenden Gewichtung. Wenn man das Ergebnis der Ableitung hat, kann man wie gewohnt das Ergebnis von der alten Gewichtung abziehen; eventuell noch gewichtet mit einer Lernrate [(Nielsen, 2016)].

Multi-layer-Perzeptrons sind wie schon erwähnt für Klassifizierungsprobleme geeignet, bei denen der Datensatz in mehr als nur zwei Klassen zu unterteilen ist. Der Backpropagation-Algorithmus bietet zudem eine effiziente Lern-Lösung selbst für große neuronale Netze. Deshalb verwendet man multi-layer-Perzeptrons heute sehr vielfältig für typische Klassifikationsprobleme, wie die Erkennung von handschriftlichen Buchstaben oder Zahlen. Hierbei geht man meist in folgenden Schritten vor:

1. Erstelle eine große Sammlung unterschiedlichster handschriftlicher Zeichen (meist mehrere 10.000)
2. Normalisiere alle Zeichen dieser Sammlung (in Größe, Orientierung, usw.) und erstelle geeignete Graustufen-Bilder
3. Erstelle ein neuronales Netz mit n Input-Neuronen (wobei n die Anzahl der Pixel in jedem Bild darstellt), m Output-Neuronen (m ist die Anzahl der Zeichen im Alphabet um das es geht, also die Klassen) und beliebig vielen Hidden-Schichten und Hidden-Neuronen.
4. Propagiere jedes Bild aus dem Trainingsset durch das neuronale Netzwerk, berechne den Fehler, propagiere den Fehler zurück in das Netzwerk und passe die Gewichtungen an. Dabei sind die Reize an den Input-Neuronen die jeweiligen Graustufenwerte der Pixel
5. Wiederhole diesen Vorgang beliebig oft aber viel

Am Ende dieser Trainingsphase sollte das Netzwerk soweit gelernt sein, dass es mit großer Wahrscheinlichkeit handschriftliche Zeichen erkennen kann und nur noch wenige Fehler produziert. Das hängt natürlich von den gewählten Hyper-Parametern (Lernrate, Anzahl Hidden-Layer, ...) und vor allem von der Anzahl der Trainings-Durchläufe (*epochs*) ab [(Nielsen, 2016)].

5 Gegenwärtige Entwicklungen und Ausblick

Heutige künstliche neuronale Netze sind sehr vielfältig in Ansatz, Architektur und Anwendungsgebieten. Das grundlegende Prinzip einer vernetzten Struktur mit Schwellenwerten und Aktivitätsfunktionen ist aber durchaus erhalten geblieben. Mit wachsender Rechenleistung steigt die Effizienz der Lernalgorithmen natürlich an; gleichzeitig stellt

man KNNs auch vor größere Herausforderungen. Die Objekterkennung auf Bildern mit teilweise sehr hochfrequenten Bildabschnitten ist eine solche. Die Vorverarbeitung ist in vielen Fällen nur eingeschränkt möglich (Farbkanäle extrahieren, Kantendetektion usw.) und führt nicht unbedingt zum gewünschten Ziel.
Man ist mittlerweile dazu übergegangen, maschinelles Lernen wieder Richtung „richtiger“ künstlicher Intelligenz zu lenken, die in erster Linie nicht auf einen externen Lehrer angewiesen ist, sondern sich Daten-Features automatisch extrahiert. Dies kommt näher an die Wirklichkeit heran, bei der Menschen und Tiere durch Beobachtung und Erfahrung lernen, anstatt dadurch, dass sie bei jedem Objekt erklärt bekommen, wie es heißt. Ein solches Bestreben fällt in das Forschungsgebiet des *deep learning* [(LeCun et al., 2015)].

Als Beispiel kann man hier die sogenannten *deep convolutional nets* nennen, bei denen es nicht darum geht, ein Netz mit der Angabe von erwünschten Werten zu lernen, sondern ein Schichtensystem zu erstellen, bei denen aus einem Bild ein immer abstrakteres Set von Features automatisch extrahiert wird. Angefangen mit der untersten Ebene, der Rohpixel-Ebene folgen Ebenen, die eine rudimentäre Kantendetektion vornehmen. Höhere Ebenen versuchen die Kanten Objekten zuzuordnen und so immer abstraktere Zusammenhänge zu finden. Es wird noch einige Forschung in diesem Gebiet getätigt werden müssen, aber die bisherigen Ergebnisse sind vielversprechend [(LeCun et al., 2015)].

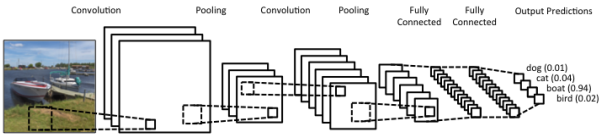


Abbildung 10: Die Architektur eines convolutional neural network.

6 Zusammenfassung und Fazit

Wir haben gesehen, welchen Wandel künstliche neuronale Netze von Beginn bis in die heutige Zeit durchlebt haben und welche Konzepte sich mittlerweile durchgesetzt haben. Das Gebiet der KNNs wächst auch heute noch stetig, nicht zuletzt mit der Entwicklung in Richtung unsupervised deep learning. Neben anderen Bestrebungen in der Problemklasse der Klassifizierungen wie zum Beispiel den Support-Vector-Machines setzen sich KNNs sehr gut durch, nicht zuletzt auch wegen der Anschaulichkeit und wegen des Traums, ein künstliches Gehirn zu schaffen.

Literatur

Yves Chauvin and David E. Rumelhart. 1995. *Backpropagation: Theory, Architectures, and Applications.* Lawrence Erlbaum Associates, Hillsdale, New Jersey, USA.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521 (2015), 436–444.

Warren McCulloch and Walter Pitts. 1943. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics* 5 (1943), 115–133.

Michael Nielsen. 2016. Neural Networks and Deep Learning. (2016). `http://neuralnetworksanddeeplearning.com`

springer.com. o.J. Neuronen und Gliazellen. (o.J.). `www.springer.com/cda/content/document/.../978-3-8274-2028-2_c2.pdf`

uni due.de. o.J. Definition Lernen. (o.J.). `https://www.uni-due.de/edit/lp/common/lernen.htm`

uni freiburg.de. o.J. Rosenblatt's Perzeptron-Algorithmus. (o.J.). `http://ml.informatik.uni-freiburg.de/_media/documents/teaching/ss09/ml/perceptrons.pdf`

uni muenster.de. o.J. Geschichte der künstlichen neuronalen Netze. (o.J.). `http://wwwmath.uni-muenster.de:8010/Professoren/Lippe/lehre/skripte/wwwnnscript/ge.html`

Abbildungsverzeichnis