Machine Learning 3: Neuronale Netze II

Ergänzungsfach Informatik, 2021/2022, pro@kswe.ch

19. November 2021

Künstliche neuronale Netze basieren auf der Modellierung von Neuronen. Das Modell eines Neurons wurde durch die Biologie motiviert. Bereits in den 1950er-Jahren versuchte man dem Computer dadurch das Lernen beizubringen, in dem man die Arbeitsweise von Neuronen simulierte und dadurch versuchte Probleme zu lösen. Nach einigen Tiefschlägen haben die künstlichen neuronalen Netze wieder an Popularität gewonnen, da Sie erfolgreich Probleme lösen können, in dem zahlreiche künstliche Neuronen kombiniert werden. Es lohnt sich somit, einen kleinen Exkurs in das Gebiet der Biologie zu unternehmen und die Motivation für künstliche neuronale Netze zu verstehen.

1 Neuronen - die Rechenmaschine der Natur[1]

Die Gehirne von Tieren stellen für Wissenschaftler immer noch ein Rätsel dar. Selbst kleine Exemplare, wie das Gehirn einer Taube, ist weitaus leistungsfähiger als Digitalcomputer mit einer riesigen Anzahl von elektronischen Verarbeitungselementen, einem riesigen Speicherplatz und Ausführungsgeschwindigkeiten, die wesentlich höher liegen als die fleischigen, schwammigen natürlichen Gehirne.

Man hat deshalb die architektonischen Unterschiede zwischen den Gehirnen und Computern untersucht. Herkömmliche Computer verarbeiten die Daten vorwiegend sequenziell und nach ziemlich konkreten Vorschriften. Bei ihren kalten, harten Berechnungen gibt es weder Unschärfe (engl. Fuzziness) noch Mehrdeutigkeit. Dagegen verarbeiten tierische Gehirne, obwohl sie offensichtlich mit wesentlich langsameren Taktgeschwindigkeiten laufen, die Signale parallel, und Fuzziness ist ein entscheidendes Merkmal ihrer Verarbeitung.

Abbildung 1 zeigt die Grundeinheit eines biologischen Gehirns - das Neuron (auch Nervenzelle genannt).

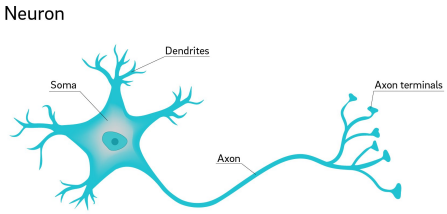

Abbildung 1: Neuron mit Dendriten, Axon und Terminalen.

Es gibt zwar verschiedene Formen von Neuronen, doch übertragen sie alle ein elektrisches Signal von einem Ende zum anderen, von den Dendriten entlang der Axone zu den Terminalen. Diese Signale werden dann von einem Neuron an ein anderes übergeben. Auf diese Weise nehmen wir Licht, Töne, Druck, Wärme usw. wahr. Signale von spezialisierten Sensorneuronen werden entlang des Nervensystems zum Gehirn übertragen, das selbst wieder aus Neuronen besteht.

Wie viele Neuronen brauchen wir, um interessantere, komplexere Aufgaben zu realisieren? Das sehr leistungsfähige menschliche Gehirn enthält ungefähr 100 Milliarden Neuronen! Eine Fruchtfliege besitzt lediglich 100 000 Neuronen und ist damit schon in der Lage, zu fliegen, zu fressen, Gefahren auszuweichen, Nahrung zu suchen und viele andere ziemlich komplexe Aufgaben zu erledigen. Da die Anzahl von 100 000 Neuronen im Kapazitätsbereich moderner Computer liegt, könnte man doch

[1] Auszug aus dem Buch "Neuronale Netze selbst programmieren, Tariq Rashid"

versuchen, ein solches Gehirn nachzubilden. Ein Fadenwurm hat nur 302 Neuronen, was geradezu verschwindend gering ist, verglichen mit den Ressourcen heutiger Digitalrechner! Doch dieser Wurm kann einige recht nützliche Aufgaben bewältigen, mit denen herkömmliche Computerprogramme von viel grösserem Umfang nicht zurechtkämen.

Worin liegt also das Geheimnis? Warum sind biologische Gehirne so leistungsfähig, wenn man bedenkt, dass sie - verglichen mit modernen Computern - wesentlich langsamer sind und aus relativ wenigen Verarbeitungselementen bestehen? Die vollständige Funktionsweise von Gehirnen, wie zum Beispiel des Bewusstseins, ist immer noch ein Geheimnis. Doch weiss man inzwischen genügend über Neuronen, um auf verschiedenen Arten der Verarbeitung schliessen zu können.

Sehen wir uns also an, wie ein Neuron funktioniert. Es übernimmt ein elektrisches **Eingangssignal** und gibt ein **anderes elektrisches Signal aus**. Dies erinnert stark an das EVA-Prinzip: Das Eingangssignal ist die Eingabe, das Signal wird verarbeitet und die Ausgabe ist ein Ausgabesignal. Könnten wir Neuronen als lineare Funktionen darstellen, wie wir es zuvor schon getan haben? Dies funktioniert leider nicht gut. Ein biologisches Neuron produziert keine Ausgabe, die schlichtweg eine einfache lineare Funktion der Eingabe ist. $y = m \cdot x + b$ ist also zu simpel.

Beobachtungen legen nahe, dass Neuronen nicht sofort reagieren, sondern stattdessen die Eingabe unterdrücken, bis sie ausreichen gross ist, um ein Ausgabesignal auszulösen. Man kann sich dies als **Schwellwert** vorstellen, der erreicht sein muss, bevor irgendein Ausgabesignal entsteht. Intuitiv ist das sinnvoll - die Neuronen sollen winzige Rauschsignale nicht durchlassen, sondern nur ausdrücklich starke, gewollte Signale. Eine Funktion, die das Eingangssignal übernimmt und ein Ausgangssignal generiert, dabei aber eine Art Schwellwert berücksichtigt, wird **Aktivierungsfunktion** genannt. Im mathematischen Sinne gibt es viele derartige Aktivierungsfunktionen, die diesen Effekt erzielen. Abbildung 2 zeigt ein Beispiel für eine Aktivierungsfunktion.

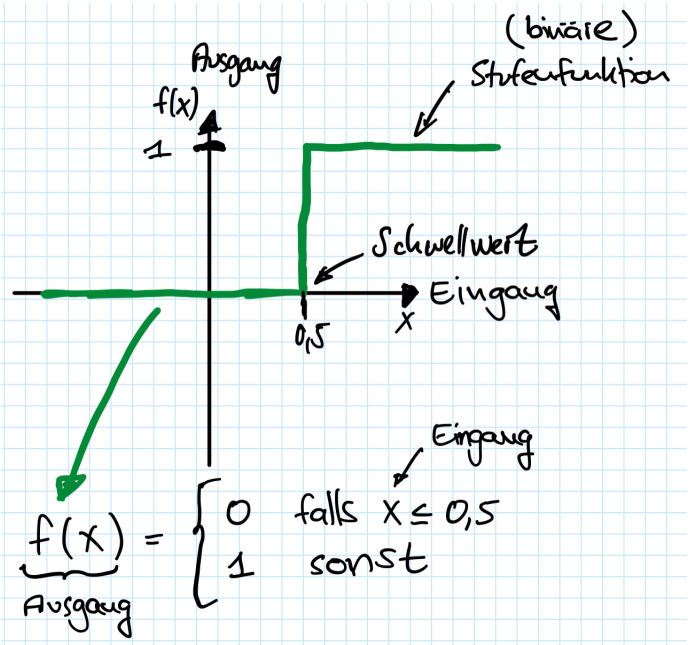

Abbildung 2: Binäre Stufenfunktion, da $f(x)$ nur zwei Werte annehmen kann - 0 und 1.

Wie aus Abbildung 2 hervorgeht, ist bei kleinen Eingabewerten die Ausgabe null. Nachdem aber die Eingabeschwelle erreicht ist, geht die Ausgabe sprungartig nach oben. Ein künstliches Neuron, das sich so verhält, wirkt fast wie ein reales biologisches Neuron. Der von Wissenschaftlern verwendete Begriff beschreibt dieses Verhalten treffend: Sie sagen, dass Neuronen **feuern**, wenn die Eingabe den Schwellwert erreicht.

Nun überlegen wir uns, wie sich damit ein künstliches Neuron modellieren lässt. Zunächst ist festzustellen, dass reale biologische Neuronen viele Eingaben und nicht nur einen einzelnen Eingabewert übernehmen. Was stellen wir mit diesen Eingaben an? Wir kombinieren sie, indem wir sie **addieren**. Die resultierende Summe geht als Eingabe an die Aktivierungsfunktion, die die Ausgabe steuert. Dies spiegelt die Arbeitsweise von realen Neuronen wider. Abbildung 3 veranschaulicht das Konzept, die Eingaben zusammenzufassen und dann die Aktivierungsfunktion auf die Summe anzuwenden.

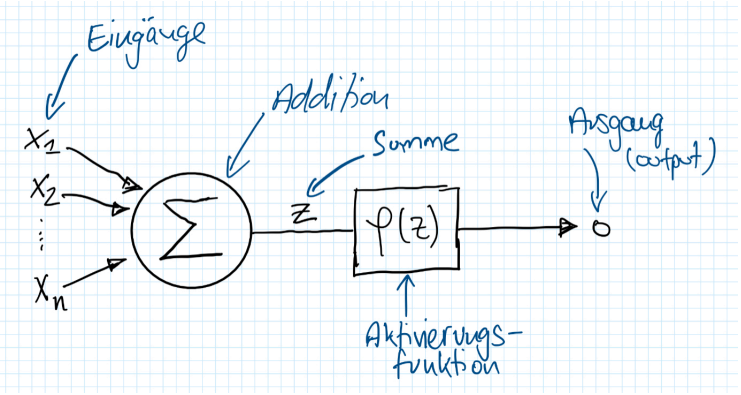

Abbildung 3: Erstes Modell für ein künstliches Neuron. Die Summe berechnet sich wie folgt: $z = x_1 + x_2 + \cdots + x_n = \sum_1^n x_i$. Mit $\varphi(z)$ wird eine Aktivierungsfunktion bezeichnet. Man kann zum Beispiel die Stufenfunktion aus Abbildung 2 verwenden. Man erhält dann $\varphi(z) = \begin{cases} 0, & \text{if } z \leq 0.5 \\ 1, & \text{sonst} \end{cases}$.

Wenn die Summe nicht gross genug ist, unterdrückt die Aktivierungsfunktion das Ausgangssignal. Ist die Summe ausreichend gross, bewirkt die Aktivierungsfunktion, dass das Neuron feuert. Interessant ist Folgendes: Wenn nur einer von mehreren Eingängen gross ist und die übrigen Eingänge lediglich einen geringen Beitrag leisten, kann das bereits genügen, damit das Neuron feuert. Darüber hinaus kann das Neuron auch feuern, wenn einige der Eingänge für sich genommen ziemlich, aber nicht ausreichend gross sind, in der Summe aber ein Signal ergeben, dass die Schwelle überwinden kann. Dies liefert einen ersten Eindruck von den komplexeren, in gewissem Sinne unscharfen Berechnungen, die derartige Neuronen realisieren können.

Als erste Erweiterung des Modells geben wir jedem Eingang ein Gewicht. Durch das Gewicht wird der Einfluss des Eingangsignals auf die Berechnung und die Aktivierung gesteuert. Ein negatives Gewicht sorgt dafür, dass das Eingangssignal die Aktivierung hemmt. Ein positives Gewicht fördert die Aktivierung. Jedes Eingangssignal erhält ein eigenes Gewicht w. Wir erhalten somit eine gewichtete Summe der Eingangssignale. Abbildung 4 zeigt das angepasste Modell.

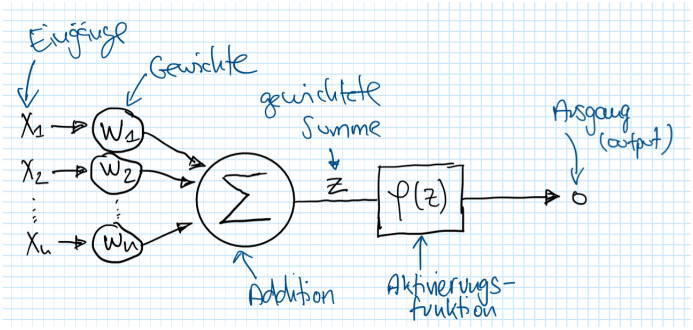

Abbildung 4: Zweites Modell für ein künstliches Neuron mit Gewichten. Die Summe berechnet sich nun wie folgt: $z = x_1 \cdot w_1 + x_2 \cdot w_2 + \cdots + x_n \cdot w_n = \sum_1^n x_i \cdot w_i$.

Wenn man die Eingänge und Gewichte als Vektoren auffasst, dann lässt sich die gewichtete Summe auch als Skalarprodukt von $\vec{x}$ und $\vec{w}$ beschreiben:

$$z = \vec{x} \circ \vec{w} = \vec{x}^T \cdot \vec{w}$$

z ist dann eine "Zahl" und kann als Eingabe für φ benutzt werden.

2 Perzeptron

Das Perzeptron ist ein Machine Learning-Verfahren für Supervised Learning mit zwei Klassen. Frank Rosenblatt stellte das Verfahren 1962 vor. Das Verfahren verwendet ein einzelnes künstliches Neuron, bei denen die Gewichte schrittweise angepasst werden.

2.1 Prinzip

Es ist ein Daten-Set $D = \{(\vec{x_1}, c_1), (\vec{x_2}, c_2), \ldots, (\vec{x_n}, c_n)\}$ gegeben. Jedes Sample besteht aus einem Feature-Vektor x_i und einem Label, die Klasse des Samples, c_i. Es gibt nur die zwei Klassen -1 und $+1$, das heisst $c_i \in \{-1, +1\}$. Besitzen die Samples n-Features (n-Dimensionen), dann gibt es $w + 1$ Gewichte. Wir möchten also den Gewichtsvektor $\vec{w} \in \mathbb{R}^{n+1}$ schrittweise durch das Perzeptron bestimmen. Das Verfahren arbeitet nun wie folgt:

1. Initialisiere die Gewichte mit zufälligen (kleinen) Werten. Füge jedem Sample den Wert 1 als zusätzliche Dimension hinzu.
2. Berechne die gewichtete Summe z, das heisst $\vec{x}_i^T \circ \vec{w}$.

   Für **jedes** Sample x_i:
3. Bestimme für $\vec{x}_i$ die vorhergesagte Klasse mit folgender Stufenfunktion:

   $$\varphi(z) = \begin{cases} +1, & \text{if } z \geq 0 \\ -1, & \text{sonst} \end{cases}$$
4. Vergleiche die vorhergesagte Klasse mit der tatsächlichen Klasse.

Fall a) Beide Klassen stimmen überein. Die Gewichte werden **nicht** angepasst.

Fall b) Beide Klassen stimmen **nicht** überein. Die Gewichte werden wie folgt angepasst:

$$\vec{w}_{neu} = \begin{cases} \vec{w}_{alt} + L \cdot \vec{x_i}, & \text{if Vorhersage } = -1 \\ \vec{w}_{alt} - L \cdot \vec{x_i}, & \text{sonst (Vorhersage } = +1) \end{cases}$$

Die Lernrate L kann auf $0,5$ gesetzt werden. Die Schritte 3 und 4 werden für alle Samples einmal ausgeführt. Man sollte auch hier wieder in Trainingsdaten und Testdaten aufteilen und nachher das erlernte Verfahren testen. Man kann die Schritte 3 und 4 auch **mehrmals** auf die gesamten Trainingsdaten anwenden. Ein Durchlauf aller Trainingsdaten wird als **Epoche** bezeichnet. Das Perzeptronverfahren benötigt eventuell mehrere Epochen bis eine Lösung gefunden wird. Man kann beweisen, dass das Verfahren immer eine Lösung findet, wenn die Daten linear separierbar sind. Es braucht eventuell einfach mehrere Epochen.

Geometrisch kann man das Verfahren im $2-D$ wie in Abbildung 5 deuten. Das Perzeptron versucht schrittweise eine Gerade zu ermitteln, welche die beiden Klassen trennt.

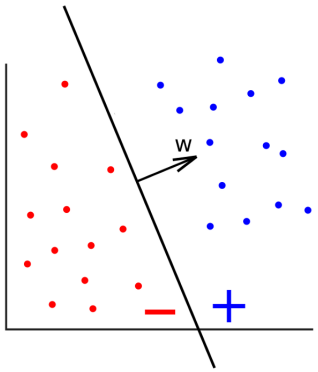

Abbildung 5: Der Vektor $\vec{w}$ definiert die Trennlinie.

Das Skalarprodukt $\vec{x}_i^T \circ \vec{w}$ definiert eine Hyperebene in Koordinatenform:
$w_1 \cdot x_{i1} + w_2 \cdot x_{i2} + \cdots + w_n \cdot x_{in} + w_0 \cdot 1 = 0$.

Lösungsvorschlag: Sie finden die Lösung in der Python-Datei `perzeptron_1.py` bzw. `perzeptron_2.py`.