9 Tiefe Architekturen

Es gibt zwei Fragen, dir durchaus unabhängig voneinander behandelt werden können, nämlich:

1. Welche Funktionen können wir mit neuronalen Netzen repräsentieren?
2. Welche Funktionen können wir mittels Datensätzen "lernen"?

Um das zu sehen, bedenke man folgendes: es kann gut sein, dass wir eine bestimmte Funktion f mit einem Netz N mit i Schichten berechnen können, aber egal mit welchen Daten wir ein zufällig initialisiertes Netz mit i Schichten trainieren, es kommt niemals das Netz heraus, das f berechnet. Man muss also die beiden Probleme separat behandeln.

In diesem Abschnitt geht es also erstmal darum, welche Art von Funktion neuronale Netze überhaupt berechnen. Neuronale Netze sind im Prinzip nur eine Art, eine Klasse von Funktionen zu schreiben, die vielleicht am anschaulichsten ist, aber nicht um bedingt am besten geeignet, die Funktion zu verstehen und zu berechnen. Neuronale Netze versteht man am besten, wenn man sie als eine Abfolge von Schichten auffasst, wobei am Ende jede Schicht nacheinander appliziert wird (wir schauen uns das später im Detail an).

Eine Netz der Tiefe 1 berechnet folgende Funktion: sei $\mathbf{a}$ der Eingabevektor, M eine Matrix, $\mathbf{b}$ ein Vektor der passenden Länge, und g eine (nichtlineare) Aktivierungsfunktion.

Ein Netz N der Tiefe 1 berechnet nun die Funktion

$$N(\vec{x}) = g(M\vec{x} + \vec{b}) \tag{111}$$

Nun nehmen wir an, wir haben ein Netz N mit Schichten $N_1, N_2, ..., N_i$. Dann berechnet N die Funktion

$$N(\vec{x}) = N_i \circ ... \circ N_1(\vec{x}) = g_i(M_i(...g_1(M_1\vec{x} + \vec{b_1})...) + \vec{b_i}) \tag{112}$$

Das heißt: wir applizieren nacheinander die einzelnen Schichten. Wichtig ist hierbei: da lineare Funktionen unter Komposition geschlossen sind, würden die Schichten zu einer einzigen kollabieren, *wenn es nicht dazwischen die nonlinearen Funktionen gäbe.* Die Mächtigkeit von tiefen Netzen besteht also in der Alternation von linearen und nicht-linearen Schichten.

Parameter und Freiheitsgrade Hierbei ist folgendes wichtig: der Parameter, der die Anzahl der Zeilen der inneren Matrizen festlegt, hat erstmal keinerlei Bedeutung für die Natur der Ausgabe, sondern ist vielmehr wichtig für die *inneren* Berechnungen. Hierfür müssen wir nochmal anschauen, was Matrizen machen: wenn wir

$$M \in \mathbb{R}^{m \times n}$$

auf einen Vektor $\vec{x} \in \mathbb{R}^n$ applizieren, dann haben wir m *unabhängige* lineare Funktionen auf $\vec{x}$ appliziert, und die m Ergebnisse liefern uns den Ausgabevektor

$$M\vec{x} \in \mathbb{R}^m$$

Das bedeutet: je größer unsere Matrizen, desto mehr unabhängige lineare Operationen können wir ausführen, und desto mächtiger werden unsere Rechenmöglichkeiten. Man nenn daher den Parameter m auch den **Freiheitsgrad** des Models. Wir haben also ein Model

$$(\vec{x} \in \mathbb{R}^n) \overset{(M_1 \in \mathbb{R}^{m_1 \times n})}{\longrightarrow} (\vec{y}_1 \in \mathbb{R}^{m_1}) \xrightarrow{g_1} (\vec{y}_2 \in \mathbb{R}^{m_2}) ....$$

Hier sind m_1, m_2 etc. die Freiheitsgrade der Funktion und Hyperparameter, die nicht durch externe Datenformate vorgegeben sind, sondern empirisch optiminert werden müssen. Allgemein gilt auch hier:

Je kleiner m_1, m_2 etc., desto enger werden unsere Generalisierung gezogen ($\cong$ weniger Gefahr von *overfitting*), aber auch: desto weniger mächtig sind unsere Modelle ($\cong$ mehr Gefahr von *underfitting*)

10 Universelle Approximation

10.1 Mächtigkeit ist nicht Lernbarkeit

Wir kommen jetzt zum sog. universal approximation theorem, dass besagt, dass wir jede berechenbare Funktion mit einem Netzwerk approximieren können. Approximieren besagt hier soviel wie: die Differenz der beiden Funktionen wird vernachlässigbar klein. Das ist natürlich ein sehr wichtiges Ergebnis, denn es besagt dass unsere Netze praktisch universell sind, also alle möglichen Funktionen darstellen können.

Satz 3 *Für jede Borel-messbare Funktion $f : \mathbb{R}^n \to \mathbb{R}^m$, jedes $\epsilon > 0$, gibt es ein neuronales Netz N mit einem hidden layer, so dass für alle $\vec{z} \in \mathbb{R}^n$ gilt:*

$$\|f(\vec{z}) - N(\vec{z})\| \leq \epsilon$$

Eine Funktion f ist Borel-messbar (etwas vereinfacht), falls das Urbild einer abzählbaren Vereinigung von geschlossenen Intervallen wiederum eine abzählbare Vereinigung geschlossener Intervalle ist.

Das besagt also: wir können bereits mit einem hidden layer *jede* solche Funktion beliebig approximieren! Allerdings ist dieses Ergebnis mit großer Vorsicht zu genießen: denn es sagt uns zwar etwas über Mächtigkeit, aber nichts darüber, wie wir die Funktionen *lernen* können. Zweitens sagt uns der Satz etwas über die Zahl der hidden layer, aber nichts über deren Größe: in der Tat sagt ein wichtiges Ergebnis, dass wir zwar die Zahl der hidden layer immer reduzieren können (bis auf 1), aber die Größe der layer wächst dabei exponentiell – das Vorgehen verbietet sich also in der Praxis.

Es gibt also insbesondere zwei Dinge zu unterscheiden:

1. Was Klassen von Funktionen *können*, und
2. wie Funktionen lernen.

Tiefe Netze sind insbesondere wichtig, denn je tiefer das Netz, desto besser sind die Generalisierungen, die Netze ziehen (für komplexe Aufgaben).

Man kann das sich wie folgt klarmachen: nehmen wir an, wir möchten eine Boolesche Funktion

$$B : \{0,1\}^n \to \{0,1\}$$

Es gibt viele solche Funktionen, nämlich

$$2^{2^n}$$

Wenn wir nun eine Funktion mit einem neuronalen Netz lernen möchten, das *ein* hidden layer hat, dann brauchen wir mindestens 2^n Zellen, und es ist schwierig, damit aus den Eingaben eine relevante Generalisierung zu ziehen. Das sieht ganz anders aus, wenn wir mehrere Schichten haben: dann kann z.B. Boolesche Entscheidungsbäume (bis zu einer gewissen Tiefe) simulieren und damit sehr schöne Generalisierungen treffen.

10.2 Ein Beispiel

Boolesche Funktionen kann man sehr schön nutzen, um die Art und Weise, wie neuronale Netze funktionieren, zu illustrieren. Nehmen wir die bekannte XOR-Funktion. Unsere Eingaben sind Vektoren in $\{0,1\}^2$. Wir machen nun die erste (lineare) Schicht:

$$M_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{pmatrix}$$

Damit bekommen wir:

$$M_1(1,0) = (1,0,-1,0) \tag{113}$$
$$M_1(0,1) = (0,1,0,-1) \tag{114}$$
$$M_1(1,1) = (1,1,-1,-1) \tag{115}$$
$$M_1(0,0) = (0,0,0,0) \tag{116}$$

Das transformiert also die Eingabe. Wir können nun eine *maxout*-unit als nichtlineare Transformation nehmen, mit $G_1 = \{1,2\}$, $G_2 = \{3,4\}$. So bekommen wir:

$$g \circ M_1(1,0) = (1,0) \tag{117}$$
$$g \circ M_1(0,1) = (1,0) \tag{118}$$
$$g \circ M_1(1,1) = (1,-1) \tag{119}$$
$$g \circ M_1(0,0) = (0,0) \tag{120}$$

Es ist nicht schwer zu sehen, dass die Vektoren nun linear separierbar sind entlang der gewünschten Grenze. Das sagt uns, dass wir bereits mit einem linearen Modell auskommen. Wir nehmen eine sehr einfache Matrix:

$$M_2 = \begin{pmatrix} 1 & 1 \end{pmatrix}$$

Natürlich gilt hier einfach: $M_2(x_1, x_2) = x_1 + x_2$, also:

$$M_2 \circ g \circ M_1(1,0) = 1 \tag{121}$$
$$M_2 \circ g \circ M_1(0,1) = 1 \tag{122}$$
$$M_2 \circ g \circ M_1(1,1) = 0 \tag{123}$$
$$M_2 \circ g \circ M_1(0,0) = 0 \tag{124}$$

An dieser Stelle ist eine weitere nicht-lineare Funktion unnötig, unser Netz berechnet bereits die XOR-Funktion.

10.3 Allgemeiner: Entscheidungsbäume

Jede Boolesche Funktion

$$B : \{0,1\}^n \to \{0,1\}$$

lässt sich mittels eines binären Entscheidungsbaumes mit Tiefe $\leq n$ darstellen. Der Witz an dieser Darstellung ist, dass wir oftmals nicht den vollen Entscheidungsbaum der Tiefe n brauchen, sondern nur ausnahmsweise, z.B. für Funktionen wie

$$B_{ger}(a_1, ..., a_n) = 1 \Leftrightarrow a_1 + ... + a_n \text{ gerade} \tag{125}$$

Sonst reicht es oftmals, eine bestimmte Kombination von z.B. 3 Merkmalen zu kennen (aus z.B. 20) um bereits eine Entscheidung zu treffen. Die Konstruktion von guten Entscheidungsbäumen basiert dabei darauf, dass wir die wichtigsten/informativsten Merkmale zuoberst setzen, und die unwichtigsten zuunterst. Hiermit können wir insbesondere in der Praxis gute Generalisierungen treffen. Wohlgemerkt: das muss nicht so sein, ist es aber in der Praxis, hängt also ab von der Klasse von Problemen, die wir betrachten.

Diese Beobachtung kann man nun auf neuronale Netze übertragen: ein Netz mit einem hidden layer wird als solches keine Boolesche Funktion vor einer anderen bevorzugen: erstmal stehen alle auf derselben Stufe. Wenn wir ein Netz mit n hidden layern haben, dann können wir uns das vorstellen als einen (viel mächtigeren) Entscheidungsbaum mit n Ebenen. Mächtiger, denn wir können auf jeder Ebene viel mehr machen als ein Entscheidungsbaum, aber: jede Generalisierung des Entscheidungsbaumes lässt sich eben auch auf

der Ebene eines layers fassen. Wenn nun ein Merkmal M_i besonders wichtig ist, dann kann also der Unterschied zwischen

$$B_{M_i=0} \tag{126}$$

$$B_{M_i=1} \tag{127}$$

sehr weit oben im Netz gezogen werden. Nimm hierzu die Eingabe

$$\vec{x} = (x_1, ..., x_{i-1}, 0, x_{i+1}, ...x_n) \tag{128}$$

und das lineare Modell mit

$$M = \begin{pmatrix} x_{1,1} & x_{1,2} & ...x_{1,i} & .... & x_{1,n} \\ .... & & & & \end{pmatrix} \tag{129}$$

wobei wir setzen:

$$x_{1,1} = x_{1,2} = ... = x_{1,i-1} = x_{1,i+1} = ... = x_{1,n} = 0 \tag{130}$$

$$x_{1,i} = 1 \tag{131}$$

In diesem Fall wird der gesamte Wert der ersten Ausgabezeile 0 gdw. $x_i = 0$, während die restlichen Zeilen andere Merkmale filtern. Eine nichtlineare *maxout*-unit erlaubt uns, verschiedene Gruppen in Abhängigkeit zu setzen, z.B. durch

$$\vec{z}_j = max(y_1, y_k) \tag{132}$$

Hier bekommen wir also z.B. den Wert 1, falls $x_{1,i} = 1$ (je nach herangehensweise). Die nächst Schicht prüft das zweitwichtigste Merkmal etc.

Der Trick ist hier: dadurch, dass wir Berechnungen, die wir auch in einem layer durchführen könnten, auf verschiedene layer verteilen, führt unser Lernalgorithmus zur schnelleren Optimierung (gradient descent) gdw. wir richtig generalisieren. Eine starke Berücksichtigung des wichtigsten Merkmals im ersten layer wird schneller die Ergebnisse verbessern, und daher wird es eher erfolgen (wg. Gradienten!). Hier sehen wir die Herausforderung für Lernalgorithmen in tiefen Netzen: wir wollen zuerst allgemeinere Generalisierungen in früheren Schichten ziehen. Es ist aber mathematisch nicht garantiert, dass das einfach funktioniert, es ist nur konzeptuell einleuchtend!

Das führt uns also zum schwierigen Problem von Training und Optimierung neuronaler Netze.