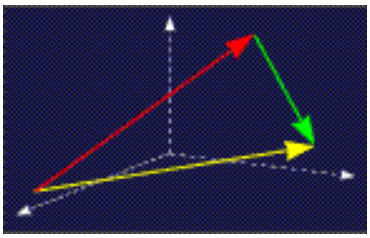
mathe online
Skripten
http://www.mathe-online.at/skripten/



Funktionen, die im Deep Learning auftreten, und deren Gradienten

Franz Embacher

Fachhochschule Technikum Wien | Fakultät für Mathematik der Universität Wien
E-mail: embacher@technikum-wien.at
WWW: http://homepage.univie.ac.at/franz.embacher/

Dieses Skriptum informiert über einige mathematische Aspekte, die für moderne Verfahren des Machine Learning, insbesondere für das Deep Learning, grundlegend sind. Es versteht sich *nicht* als Einführung in diese Gebiete, sondern will vorbereitend wirken, indem der Begriff des neuronalen Netzes (in einer einfachen Variante) aus einem mathematischen Blickwinkel heraus entwickelt und beleuchtet wird. Im Zentrum stehen dabei Funktionen in mehreren (tatsächlich *sehr vielen*) Variablen, die typischerweise im Deep Learning auftreten, und deren Gradienten. Voraussetzungen sind: Grundkenntnisse der linearen Algebra (lineare Abbildungen von $\mathbb{R}^n$ nach $\mathbb{R}^m$, Matrizenrechnung) und der Analysis in mehreren Variablen (Gradient, Kettenregel in mehreren Variablen und eine Ahnung, was das Gradientenabstiegsverfahren ist).

1 Lineare Abbildungen

Sie kennen bereits das Konzept der **linearen Abbildungen** von $\mathbb{R}^n$ nach $\mathbb{R}^m$. Es handelt sich dabei um genau jene Abbildungen $\mathbb{R}^n \to \mathbb{R}^m$, die in der Form

$$x \mapsto Wx \tag{1.1}$$

geschrieben werden können, wobei W eine $m \times n$-Matrix ist (also $W \in \mathbb{R}^{m\times n}$) und das Argument $x \in \mathbb{R}^n$ als Spaltenvektor mit n Komponenten aufgefasst wird. Wx steht demnach für das Produkt der Matrix W mit dem Vektor x. Die k-te Komponente von Wx ist gegeben durch

$$(Wx)_k = \sum_{l=1}^{n} W_{kl} x_l \qquad \text{für } k = 1, \ldots, m, \tag{1.2}$$

wobei W_{kl} die kl-Komponente der Matrix W und x_l die l-te Komponente des Vektors x ist. Mit der Abkürzung

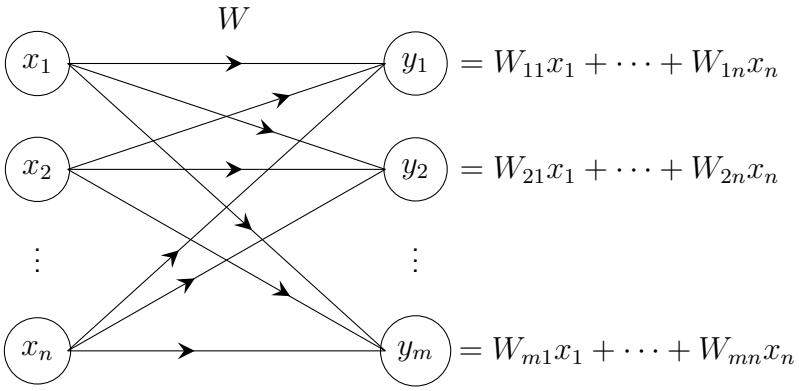
$$y = Wx \tag{1.3}$$

kann man die Zuordnungsvorschrift (1.1) auch in der Form $x \mapsto y$ mit

$$y_k = \sum_{l=1}^{n} W_{kl} x_l \qquad \text{für } k = 1, \ldots, m. \tag{1.4}$$

schreiben. Das Symbol W bezeichnet also die Matrix, in der die Koeffizienten der Zuordnungsvorschrift $x \mapsto Wx$ zusammengefasst sind. Wir wollen das gleiche Symbol auch für die Zuordnungsvorschrift selbst verwenden – diese beiden Konzepte sind so eng miteinander verbunden, dass wir zwischen ihnen nicht unterscheiden müssen.

Eine lineare Abbildung $W : \mathbb{R}^n \to \mathbb{R}^m$ kann auch als (sehr einfaches, lineares) künstliches neuronales Netz dargestellt werden – ein eher unorthodoxer Zugang, der aber im Kontext des Machine Learning sinnvoll ist, weil er das Verständnis für die in diesem Gebiet verwendeten mathematischen Techniken erleichtert. Dazu stellen wir uns vor, dass der Vektor x aus n reellen Zahlen besteht, die gemeinsam irgendein Stück Information darstellen. Aus diesen wollen wir auf lineare Weise m reelle Zahlen gewinnen, die in einem Vektor y zusammengefasst werden. Auch die Komponenten des Vektors y stellen zusammen ein Stück Information dar, das aus x gewonnen wird. Nun betrachten wir die folgende Grafik:



Links sehen wir Platzhalter für die n Zahlen $x_1, x_2, \ldots, x_n$, rechts Platzhalter für die m Zahlen $y_1, y_2, \ldots, y_m$. Jeder Kreis stellt – gemäß einer *sehr* vereinfachten Sicht eines Gehirns – ein künstliches „Neuron“ dar. Dieses neuronale Netz besteht aus zwei „Schichten“ (*layer*), der Eingabeschicht, in die konkrete Werte für die Komponenten des Vektors x eingespeist werden, und der Ausgabeschicht, in der dann die Komponenten des 5 x abhängigen Vektors y stehen. Dass die y_k linear von den x_l abhängen, kommt durch die in der Grafik angegebenen Formeln für die y_k, die gemeinsam genau der Zuordnungsvorschrift (1.4) entsprechen, zum Ausdruck: Die auftretenden Koeffizienten sind gerade die Komponenten der Matrix W. Jedes y_k hängt von all jenen x_l ab, für die $W_{kl} \neq 0$ ist. In der Sprache der neuronalen Netze können wir formulieren, dass das l-te Neuron der Einhabeschicht „feuert“, indem es den Neuronen der Ausgabeschicht Signale in Form von Zahlen zuschickt. Dabei bekommt das k-te Neuron der Ausgabeschicht vom l-ten Neuron der Eingabeschicht nicht einfach den Eingabewert x_l zugeschickt, sondern

das Produkt $W_{kl}x_l$. Der Eingabewert x_l wird also gewissermaßen mit dem „Gewicht“ (*weight*) W_{kl} versehen. (Vom englischen *weight* kommt auch der Name W unserer Matrix.) Beim k-ten Neuron der Ausgabeschicht werden die empfangenen Signale addiert, wodurch sich die Ausgabewerte

$$y_k = W_{k1}x_1 + \cdots + W_{kn}x_n = \sum_{l=1}^{n} W_{kl}x_l \tag{1.5}$$

ergeben. Hier haben wir wieder die Formel (1.4). Die Pfeile in der Grafik symbolisieren diese Abhängigkeiten der y_k von den x_l. (Sind alle $W_{kl} \neq 0$, so können wir sagen, dass *jedes* y_k von *allen* x_l abhängt. Dem entsprechen insgesamt mn Pfeile (von denen in der Grafik neun dargestellt sind). Jeder Pfeil entspricht somit einer Komponente der Matrix W. Insofern kann man die Koeffizienten W_{kl} als Gewichte der Verbindungen zwischen den Neuronen ansehen. Das drückt die Beschriftung W in der Grafik (oberhalb der Pfeile) aus.

Wir wollen die Frage, wie zulässig der Begriff „Neuron“ ist, nicht auf die Goldwaage legen. Ein biologisches Neuron empfängt Signale und leitet Signale weiter. Wir werden später Netze betrachten, die aus mehr als nur zwei Schichten bestehen und damit einem biologischen neuronalen Netz näher kommen. Generell wollen wir alle durch Kreise gekennzeichneten Bestandteile der Schichten in einem solchen Diagramm als Neuronen bezeichnen.

2 Affine Abbildungen

Wir verallgemeinern nun das bisher entwickelte Konzept eines neuronalen Netzes ein bisschen. Dabei halten wir daran fest, dass m Ausgabewerte y_k von n Eingabewerten x_l abhängen, verändern aber die Form dieser Abhängigkeit: Wir erlauben, dass nach der Anwendung einer linearen Abbildung noch ein vorgegebener Vektor (mit m Komponenten) zum Ergebnis addiert wird. Damit landen wir bei den **affinen Abbildungen**. Eine affine Abbildung von $\mathbb{R}^n$ nach $\mathbb{R}^m$ ist von der Form

$$x \mapsto Wx + b, \tag{2.1}$$

wobei W eine reelle $m \times n$-Matrix und b ein reeller m-Vektor ist, also $W \in \mathbb{R}^{m\times n}$ und $b \in \mathbb{R}^m$. Für gegebenes W und b bezeichnen wir die zugehörige affine Abbildung mit $\mathcal{A}_{W,b}$:

$$\begin{aligned} \mathcal{A}_{W,b} &: \mathbb{R}^n \to \mathbb{R}^m \\ \mathcal{A}_{W,b}(x) &= Wx + b. \end{aligned} \tag{2.2}$$

In Komponenten angeschrieben, wirkt diese Abbildung so:

$$\mathcal{A}_{W,b}(x)_k = \sum_{l=1}^{n} W_{kl}x_l + b_k \qquad \text{für } k = 1, \ldots, m. \tag{2.3}$$

Mit

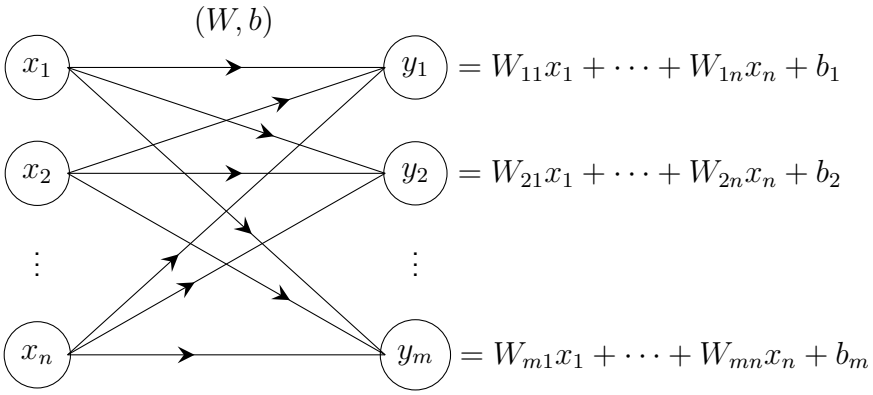
$$y = Wx + b \tag{2.4}$$

können wir das auch in der Form

$$y_k = \sum_{l=1}^{n} W_{kl}x_l + b_k \qquad \text{für } k = 1, \ldots, m \tag{2.5}$$

schreiben. (In Klammer sei hinzugefügt, dass affine Funktionen manchmal auch als „linear“ bezeichnet werden. Vielleicht wurden in Ihrem Mathematikunterricht Funktionen der Form $\mathbb{R} \to \mathbb{R}$, $x \mapsto k\,x + d$ als linear bezeichnet. In der Sprechweise der linearen Algebra ist eine solche Funktion aber nur dann linear, wenn $d = 0$ ist. Manchmal wird im Fall $d \neq 0$ der Ausdruck „linear-inhomogen“ verwendet, während im Fall $d = 0$ von einer „linear-homogenen“ Funktion gesprochen wird.)

Auch eine affine Abbildung $\mathcal{A}_{W,b} : \mathbb{R}^n \to \mathbb{R}^m$ kann als (sehr einfaches) künstliches neuronales Netz dargestellt werden:



Die affine Abbildung $\mathcal{A}_{W,b}$ wurde hier einfach mit (W, b) bezeichnet. Die Bedeutung der Komponenten des Vektors b erschließt sich, wenn wir für einen Moment annehmen, dass W die Nullmatrix ist, d.h. dass $W_{kl} = 0$ für alle k, l. Dann ist der k-te Ausgabewert $y_k = b_k$. In der Sprache der neuronalen Netze bedeutet das, dass das k-te Neuron der Ausgabeschicht, obwohl es kein Signal empfängt, sozusagen „von sich aus“ den Wert b_k annimmt. Man spricht in diesem Zusammenhang von „Verzerrung“ (*bias*, von daher rührt die Bezeichnung b). In diesem Sinn kann man b_k als Eigenschaft des k-ten Neurons der Ausgabeschicht auffassen.

3 Komponentenweise Anwendung einer Funktion

Die affinen Abbildungen bilden nur eine winzige Klasse unter jenen Funktionen $\mathbb{R}^n \to \mathbb{R}^m$, die man in Anwendungen des Machine Learning darstellen möchte. So könnte der Vektor x beispielsweise die in den Pixeln eines digitalen Bildes enthaltene Farbinformation darstellen. Für ein 1000×1000 Pixel großes Bild wären das, wenn wir 256 mögliche Werte für jede der Farben rot, grün und blau veranschlagen, 3 Millionen Komponenten des Eingabevektors x, die jeweils ganzzahlige Werte im Bereich zwischen 0 und 255 annehmen können. Vom Ausgabevektor y könnte man sich wünschen, dass er in numerischer Form angibt, welches Tier in welcher Umgebung auf dem Bild zu sehen ist. Kommen beispielsweise 50 verschiedene Tiere in Frage (etwa Hund, Katze, Eidechse, Schlange, Elefant, Wellensittich, Schaf,...) und 10 Umgebungen (etwa Wiese, Wald, felsiges Gelände, Steppe, Zoo, Wohnung,...), so sollte der Ausgabevektor y zwei Komponenten haben, eine für das Tier (mit einem der Werte $1, 2, \ldots, 50$) und eine für die Umgebung (mit einem der Werte $1, 2, \ldots, 10$). Damit hätte man 768000000 mögliche Eingabevektoren, von denen zwar nur ein kleiner Teil einem Bild mit erkennbarem Motiv entspricht, und von diesen nur ein kleiner Teil einem Bild, das eines der erlaubten Tiere in einer

der erlaubten Umgebungen zeigt, aber auch das sind noch sehr viele. Um sich nicht mit Ganzzahligkeitsbedingungen herumschlagen zu müssen, besteht der einfachste Weg darin, zunächst für x beliebige Elemente von $\mathbb{R}^{3000000}$ und für y beliebige Elemente von $\mathbb{R}^2$ zuzulassen. Die gewünschte Zuordnung $x \mapsto y$, zunächst als Abbildung $\mathbb{R}^{3000000} \to \mathbb{R}^2$ veranschlagt, sollte dann für jedes x, das ein Bild mit Tier in Umgebung darstellt, ein y liefern, dessen Komponenten jeweils einem der erlauben (ganzzahligen) Werten zumindest sehr nahekommen, und das mit der Trefferquote, mit der auch Menschen ein bestimmtes Tier in einer bestimmten Umgebung auf einem Bild erkennen. Eine solche Abbildung kann sicher nicht durch eine geschlossene Formel dargestellt werden, und es wird sich auch nicht um eine affine Abbildung handeln.

Um derartige Funktionen darstellen zu können, müssen **nicht-affine** Elemente im Spiel sein. (Statt nicht-affin sagt man manchmal auch „nichtlinear“.) Es hat sich nun herausgestellt, dass man dabei mit sehr einfachen Funktionen auskommt, und zwar mit Funktionen in nur *einer* reellen Variable, die dann zusammen mit affinen Abbildungen die Bausteine einer weitreichenden und mächtigen Methode der Funktionsapproximation bilden. Manchmal reicht sogar eine einzige Funktion $f : \mathbb{R} \to \mathbb{R}$ für eine bestimmte Anwendung, und wir wollen uns auf diesen Fall beschränken. Für f steht eine reiche Auswahl an Funktionen zur Verfügung, von denen wir nur zwei nennen wollen:

- Die ReLU-Funktion (*rectified linear unit*)

$$f(x) = \max(0, x). \tag{3.1}$$

- Die Sigmoid-Funktion

$$f(x) = \frac{1}{1 + e^{-x}}. \tag{3.2}$$

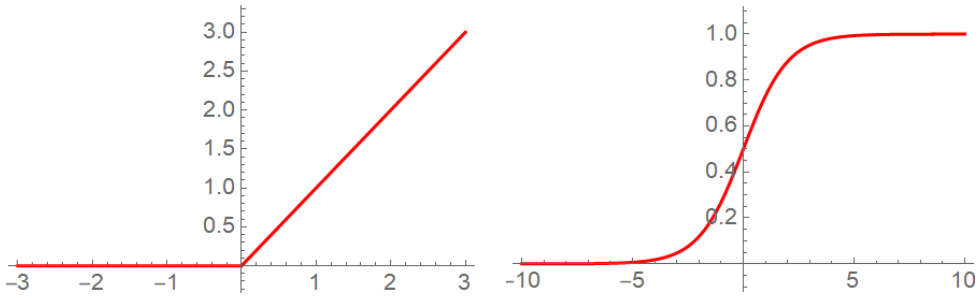


Abbildung 1: Links der Graph der ReLU-Funktion (3.1), rechts der Graph der Sigmoid-Funktion (3.2).

Welche der zur Auswahl stehenden Funktionen in einem konkreten Problem des Deep Learning verwendet wird, hängt von der Art des Problems ab und ist bis zu einem gewissen Grad Geschmackssache.

Eine solche Funktion kann, da es sich ja um eine Funktion in nur *einer* Variable handelt, zunächst nur auf Zahlen angewandt werden, nicht auf Vektoren. Um diesen Mangel zu beheben,

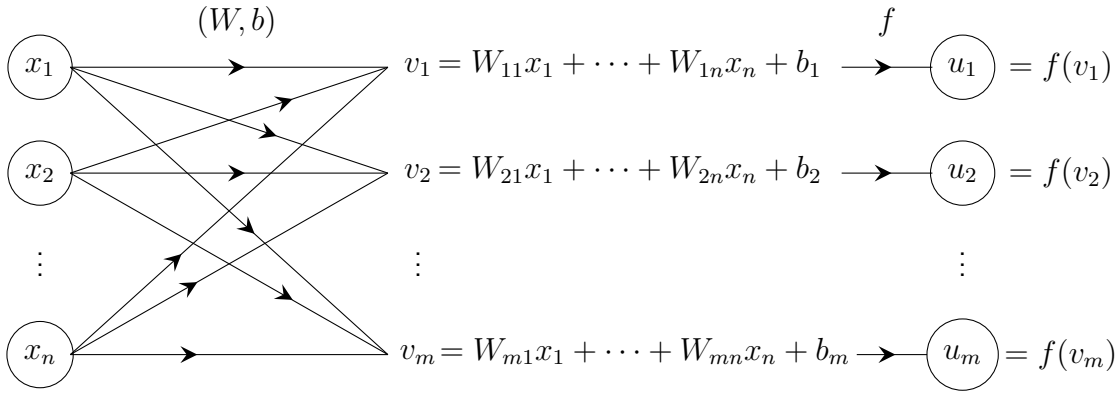
machen wir aus einer Funktion $f : \mathbb{R} \to \mathbb{R}$ eine Funktion $\mathbb{R}^n \to \mathbb{R}^n$, und zwar durch folgende Festlegung (wobei wir für die neue Funktion denselben Buchstaben f verwenden): Wirkt f auf einen Vektor x, so soll dies **komponentenweise** geschehen. Also:

$$\text{Für } \; x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \quad \text{sei } \; f(x) := \begin{pmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_n) \end{pmatrix}. \tag{3.3}$$

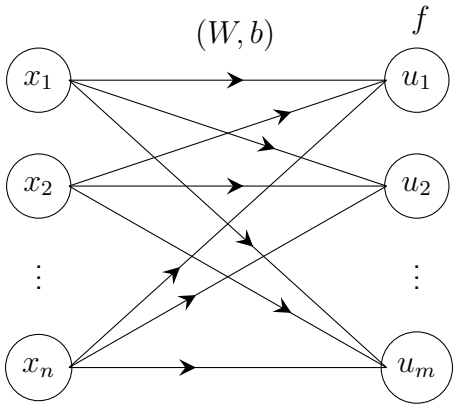
Man kann diese Regel auch in Komponenten ausdrücken:

$$f(x)_l = f(x_l) \qquad \text{für } \; l = 1, \ldots, n. \tag{3.4}$$

Ausgerüstet mit einer solchen nicht-affinen Funktion f, die komponentenweise auf Vektoren wirkt, verallgemeinern wir nun unser bisher entwickeltes künstliches neuronales Netz ein weiteres Mal, und zwar so:



Aus dem Eingabevektor x wird zunächst in einem Zwischenschritt mit Hilfe der affinen Abbildung (W, b) der Vektor $v = Wx + b$ gewonnen, und auf diesen wird die Funktion f (komponentenweise) angewandt, um den Ausgabevektor u zu erhalten. Dessen k-te Komponente u_k ist also der Wert, den das k-te Neuron der Ausgabeschicht schlussendlich erhält. Die Zwischenwerte v_k bekommen nicht den Status von Neuronen, daher sind um diese Symbole auch keine Kreise gezogen. Der Schritt von v_k zu u_k passiert gewissermaßen innerhalb des k-ten Neurons der Ausgabeschicht. Eine komparkere grafische Darstellung würde so aussehen:

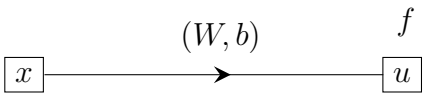


Das k-te Neuron der Ausgabeschicht nimmt seine Inputs $W_{kl}x_l$ (für $l = 1, \ldots, n$) in Empfang, bildet ihre Summe (was $(Wx)_k$ ergibt), addiert b_k und wendet danach die Funktion f an. Letztere wird in dieser Rolle **Aktivierungsfunktion** (*activation function*) genannt. Sie vervollständigt das Bild, wie künstliche Neuronen, die zu einem Netz zusammengeschlossen sind, wirken und zusammenwirken. Insgesamt stellt unser bisher entwickeltes neuronales Netz also die Abbildung

$$\begin{aligned} &\mathbb{R}^n \to \mathbb{R}^m \\ &x \mapsto f(Wx + b) \end{aligned} \tag{3.5}$$

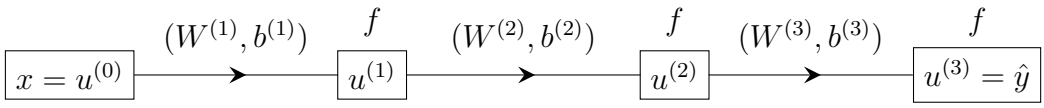
dar. Schichten von Neuronen, die auf diese Weise „kommunizieren“, sind die Elemente, aus denen neuronale Netze für realistische Anwendungen aufgebaut sind.

Wir wollen ab jetzt die Neuronen einer Schicht nicht mehr einzeln aufzeichnen, sondern stellen die ganze Schicht mit einem Rechteck-Symbol dar. Damit sieht unser neuronales Netz so aus:



4 Feedforward-Netze

Ein „tiefes“ künstliches neuronales Feedforward-Netz entsteht, wenn zumindest drei Schichten von Neuronen hintereinandergeschaltet werden, d.h. wenn es zwischen der Eingabeschicht und der Ausgabeschicht zumindest eine „verdeckte Schicht“ (*hidden layer*) gibt. Jeder Schritt von einer Schicht zur nächsten ist aufgebaut wie (3.5), wobei in aufeinanderfolgenden Schritten unterschiedliche affine Abbildungen verwendet werden, aber in der Regel immer dieselbe Aktivierungsfunktion f. (Die Bezeichnung Feedforward-Netz rührt daher, dass Information immer nur „vorwärts“ fließt. Es gibt auch allgemeinere Versionen von neuronalen Netzen, in denen Neuronen Signale innerhalb derselben Schicht oder im Sinne einer Rückkopplung an frühere Schichten weiterleiten, aber auf diese und andere Varianten von neuronalen Netzen wollen wir hier nicht eingehen.) Als Beispiel betrachten wir ein Feedforward-Netz mit zwei verdeckten Schichten, lassen aber die Anzahl der Neuronen in den einzelnen Schichten (d.h. die Dimensionen der auftretenden Matrizen und Vektoren) offen. Damit ergibt sich folgendes Szenario, zuerst grafisch dargestellt:



Ausgehend von einem Eingabevektor $x \in \mathbb{R}^n$, den wir für spätere Zwecke auch mit $u^{(0)}$ bezeichnen, werden abwechselnd affine Abbildungen und die Aktivierungsfunktion f angewandt (und zwar insgesamt dreimal). Die Ausgabe ist ein Vektor $\hat{y} \in \mathbb{R}^m$, der auch mit $u^{(3)}$ bezeichnet wird. Unser Feedforward-Netz entspricht also folgendem Kochrezept (wobei die Dimensionen der auftretenden Matrizen und Vektoren, wie bereits erwähnt, beliebig sind, solange alle Matrixoperationen durchgeführt werden können):

- Beginne mit x (gleichbedeutend mit $u^{(0)}$).
- Wähle eine Matrix $W^{(1)}$ und einen Vektor $b^{(1)}$ und wende auf x die affine Abbildung $\mathcal{A}_{W^{(1)},b^{(1)}}$ an. (Das Ergebnis werden wir später mit $v^{(1)}$ bezeichnen.)
- Wende die Funktion f komponentenweise auf das Ergebnis des letzten Schrittes an. (Das Ergebnis bezeichnen wir mit $u^{(1)}$.)
- Wähle eine Matrix $W^{(2)}$ und einen Vektor $b^{(2)}$ und wende auf $u^{(1)}$ die affine Abbildung $\mathcal{A}_{W^{(2)},b^{(2)}}$ an. (Das Ergebnis werden wir später mit $v^{(2)}$ bezeichnen.)
- Wende die Funktion f komponentenweise auf das Ergebnis des letzten Schrittes an. (Das Ergebnis bezeichnen wir mit $u^{(2)}$.)
- Wähle eine Matrix $W^{(3)}$ und einen Vektor $b^{(3)}$ und wende auf $u^{(2)}$ die affine Abbildung $\mathcal{A}_{W^{(3)},b^{(3)}}$ an. (Das Ergebnis werden wir später mit $v^{(3)}$ bezeichnen.)
- Wende die Funktion f komponentenweise auf das Ergebnis des letzten Schrittes an. Das Ergebnis ist $\hat{y}$ (gleichbedeutend mit $u^{(3)}$).

Insgesamt ist damit eine eine Abbildung $x \mapsto \hat{y}$ durch die **Verkettung** mehrerer Funktionen beschrieben. Die Matrizen $W^{(1)}$, $W^{(2)}$ und $W^{(3)}$ und die Vektoren $b^{(1)}$, $b^{(2)}$ und $b^{(3)}$ sind die **Parameter** dieser Abbildung. Sie werden oft mit dem Symbol θ zusammengefasst. Der Ausgabevektor $\hat{y}$ hängt bei gegebenen Parametern θ vom Eingabevektor x ab. Um auszudrücken, dass $\hat{y}$ von x abhängt, kann man $\hat{y}(x)$ schreiben. Um auch die Abhängigkeit von den Parametern auszudrücken, kann man $\hat{y}(x, \theta)$ schreiben, was nur die Kurzform der umständlicheren Schreibweise

$$\hat{y}(x, W^{(1)}, W^{(2)}, W^{(3)}, b^{(1)}, b^{(2)}, b^{(3)}) \tag{4.1}$$

ist (wobei jedes der W's und b's selbst wieder jeweils mehrere reelle Variable darstellt, deren Anzahl von den auftretenden Dimensionen abhängt). Die Aktivierungsfunktion f kommt nicht unter den Parametern vor, da sie in der Regel ein für allemal fix vorgegeben ist, während man die Werte der Parameter θ offen lässt, um sie der jeweiligen Aufgabe anpassen zu können. Hier sind wir – wie im Titel dieses Textes angekündigt – bei typischen Funktionen, die im Deep Learning auftreten, angelangt. Dabei wird die Abhängigkeit $\hat{y}(x, \theta)$ auf zweierlei Weise genutzt: Einerseits als die Funktion, die vom neuronalen Netz berechnet wird, also

$$x \mapsto \hat{y}(x, \theta) \qquad \text{bei vorgegebenen Parametern } \theta \tag{4.2}$$

und andererseits als Funktion, die auf die Abhängigkeit von den Parametern fokussiert, nämlich

$$\theta \mapsto \hat{y}(x, \theta) \qquad \text{bei vorgegebenem Eingabevektor } x. \tag{4.3}$$

Besonders der zweite Typ ist beim „Trainieren“ eines Feedforward-Netzes wichtig, wie im Folgenden bald klar werden wird.

Zusätzlich zu den Parametern θ und dem Eingabevektor x (aus denen sich der Ausgabevektor $\hat{y}$ ergibt) benötigen wir zunächst noch eine weitere Sache: einen Vektor y, der darstellt, was man sich eigentlich von so einem neuronalen Netz *wünscht*. Was bedeutet das? Stellen wir uns etwa vor, der Eingabevektor x stellt (wir haben dieses Beispiel bereits erwähnt) die in den Pixeln eines digitalen Bildes enthaltene Farbinformation das, und man wünscht sich die durch einen Vektor ausgedrückte Antwort auf die Frage, welches Tier in welcher Umgebung auf dem Bild zu sehen ist, als Ausgabe. Für irgendeinen erlaubten Eingabevektor x bezeichnen wir

- mit y diese korrekte Antwort (die bekannt, also vorgegeben ist)
- und mit $\hat{y}$ den Ausgabevektor des neuronalen Netzes bei einer bestimmten Wahl der Parameter θ.

Je besser das Netz seine Aufgabe erfüllt, umso näher liegt $\hat{y}$ bei y. Die große Frage besteht nun darin, wie man Parameterwerte θ findet, für die $\hat{y}$ mit y übereinstimmt oder zumindest in der Nähe von y liegt – und das nicht nur für *einen* Beispielvektor x, sondern für *viele* (also in unserem Beispiel für eine große Kollektion von Bildern, auf denen jeweils ein Tier in einer Umgebung zu sehen ist).

Das bisher entwickelte Schema verallgemeinert sich zwanglos zu Feedforward-Netzen mit einer beliebigen Anzahl an verdeckten Schichten. Wieviele man davon in einer realistischen Anwendung benötigt, ist in der Regel eine Erfahrungstatsache, muss also für verschiedene Problemtypen durch Versuche herausgefunden werden, ebenso wie die Anzahl der Neuronen in den verdeckten Schichten. Manchmal wird die Anwendung der Aktivierungsfunktion in der Ausgabeschicht weggelassen – man spricht dann von einer linearen Ausgabeschicht. Eine mathematisch hochinteressante Tatsache besteht nun darin, dass praktisch jede Funktion, die im Zusammenhang mit Machine Learning von Interesse ist, beliebig genau durch ein geeignetes Feedforward-Netz mit linearer Ausgabeschicht approximiert werden kann. Mehr dazu im letzten Abschnitt dieses Textes.

Wir kehren nun zu unserem Feedforward-Netz mit zwei verdeckten Schichten zurück und illustrieren, welche Rolle die Analysis in mehreren Variablen im Deep Learning spielt. Mit den bereits eingeführten Bezeichnungen für die Zwischenergebnisse sieht die Abfolge der Berechnungsschritte in unserem Feedforward-Netz so aus:

$$x = u^{(0)} \xmapsto{\mathcal{A}_{W^{(1)},b^{(1)}}} v^{(1)} \xmapsto{f} u^{(1)} \xmapsto{\mathcal{A}_{W^{(2)},b^{(2)}}} v^{(2)} \xmapsto{f} u^{(2)} \xmapsto{\mathcal{A}_{W^{(3)},b^{(3)}}} v^{(3)} \xmapsto{f} u^{(3)} = \hat{y} \tag{4.4}$$

Je nachdem, welche der zur Verfügung stehenden Abkürzungen verwendet werden, kann die Berechnungvorschrift auf verschiedene Weisen angeschrieben werden:

$$\begin{aligned}
\hat{y} = u^{(3)} &= f(v^{(3)}) = f\big(\mathcal{A}_{W^{(3)},b^{(3)}}(u^{(2)})\big) = f\big(W^{(3)}u^{(2)} + b^{(3)}\big) = \\
&= f\big(W^{(3)} f(v^{(2)}) + b^{(3)}\big) = f\Big(W^{(3)} f\big(\mathcal{A}_{W^{(2)},b^{(2)}}(u^{(1)})\big) + b^{(3)}\Big) = \\
&= f\Big(W^{(3)} f\big(W^{(2)}u^{(1)} + b^{(2)}\big) + b^{(3)}\Big) = \\
&= f\Big(W^{(3)} f\big(W^{(2)} f(v^{(1)}) + b^{(2)}\big) + b^{(3)}\Big) = \\
&= f\Big(W^{(3)} f\big(W^{(2)} f(\mathcal{A}_{W^{(1)},b^{(1)}}(x)) + b^{(2)}\big) + b^{(3)}\Big) = \\
&= f\Big(W^{(3)} f\big(W^{(2)} f(W^{(1)}x + b^{(1)}) + b^{(2)}\big) + b^{(3)}\Big).
\end{aligned} \tag{4.5}$$

Für jede Wahl der Parameter θ (also für jede Wahl von $W^{(1)}$, $b^{(1)}$, $W^{(2)}$, $b^{(2)}$, $W^{(3)}$ und $b^{(3)}$) gibt die letzte Zeile $\hat{y}$ explizit als Funktion von x an. Die darüber stehenden Zeilen drücken denselben Sachverhalt in unterschiedlich abgekürzter Weise aus – sie werden sich als sehr nützlich erweisen, wenn wir zur Gradientenberechnung kommen. Wieso Gradientenberechnung? Wir bitten noch um ein bisschen Geduld!

5 Kostenfunktion

Um zu messen, wie stark $\hat{y}$ vom gewünschten Ergebnis y abhängt, wird eine sogenannte Kostenfunktion $(\hat{y}, y) \mapsto L(\hat{y}, y)$ eingeführt. Ein Beispiel für eine sehr einfache Kostenfunktion wäre

$$L(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^T(\hat{y} - y) \equiv \frac{1}{2}\|\hat{y} - y\|^2. \tag{5.1}$$

Sie nimmt nur nicht-negative Werte an, ist umso größer, je weiter $\hat{y}$ und y auseinanderliegen (wobei als Entfernungsbegriff die euklidische Norm zugrunde gelegt ist) und ist genau dann gleich 0, wenn $\hat{y} = y$ ist, d.h. wenn der Ausgabevektor die richtige Antwort genau trifft. Beginnt man mit irgendeiner Wahl der Parameter θ, so wird das natürlich kaum der Fall sein. Man kann auch nicht annehmen, dass es eine immer Wahl von θ gibt, bei der für jeden Eingabevektor exakt $\hat{y} = y$ ist. Aber wir können die Frage stellen, **bei welchen Parameterwerten θ die Kostenfunktion minmal wird**. In der Praxis reicht es aus, ein lokales Minimum zu finden oder zumindest zu approximieren, sofern das Netz dann hinreichend gute Ergebnisse liefert. Und damit landen wir bei einem Optimierungsproblem. Aufgrund der großen Anzahl von Variablen kann man es nicht durch Berechnungen auf dem Papier lösen, sondern benötigt einen geeigneten Algorithmus, der das leistet. Man sagt dann, dass das neuronale Netz die Parameterwerte, die den Approximationsfehler minimieren (oder zumindest Parameterwerte, die gute Ergebnisse liefern) „lernen“ muss. Da das gewünschte Ergebnis y vorab bekannt ist, spricht man von überwachtem Lernen (*supervised learning*).

Tatsächlich will man den Approximationsfehler natürlich nicht nur für *einen* Eingabevektor x minimieren, sondern für *viele*. Man kann dann beispielsweise die Gesamt-Kostenfunktion als Mittelwert über die Kostenfunktionen für eine ganze Kollektion von Beispielvektoren x (man sagt auch: für eine Kollektion von Trainingspaaren (x, y)) definieren. Da sich dadurch mathematisch außer einer zusätzlichen Summenbildung nicht viel ändert, wollen wir nicht so weit gehen und beschränken uns wie auch bisher in diesem Text auf einen einzelnen Eingabevektor x.

6 Bedeutung des Gradienten der Kostenfunktion

Sind x und y vorgegeben, so deuten wir die Kostenfunktion als Abbildung

$$\mathcal{L} : \theta \mapsto L(\hat{y}(x, \theta), y). \tag{6.1}$$

Damit wird jedem möglichen Satz θ von Parameterwerten der zugehörige Wert der Kostenfunktion zugeordnet. Diese Funktion $\mathcal{L}$ gilt es zu minimieren. Beachten Sie, dass in (6.1) in Form des ersten Arguments $\hat{y}(x, \theta)$ eine Funktion vom Typ (4.3) auftritt! Beginnen wir nun mit irgendwelchen Parameterwerten θ_0, so lautet die Frage, wie sie zu $\theta_1 = \theta_0 + \delta\theta$ verändert (d.h. angepasst) werden können, damit $\mathcal{L}(\theta_1)$ kleiner ist als $\mathcal{L}(\theta_0)$. Ist dieser Schritt erfolgreich, so wird er mit θ_1 als Ausgangswert wiederholt, und so wird die Genauigkeit, mit der das Feedforward-Netz die korrekte Antwort y approximiert, schrittweise verbessert. Wir vertiefen uns jetzt nicht in die Details einer solchen Berechnung (insbesondere klammern wir die Frage nach der geeigneten Schrittweite $\delta\theta$ aus), sondern konzentrieren uns auf einen grundlegenden

Aspekt: Die „Richtung“ im Raum aller θ, in die man von θ_0 aus ein Stück $\delta\theta$ gehen muss, um zu $\theta_1 = \theta_0 + \delta\theta$ zu gelangen, ist $-\nabla_\theta \mathcal{L}(\theta_0)$, also minus der Gradient der Kostenfunktion im Punkt θ_0. Mit anderen Worten: Man wendet das **Gradientenabstiegsverfahren** (genauer: die eine oder andere Variante des Gradientenabstiegsverfahrens) in der – in der Regel ziemlich hochdimensionalen – Variablen θ an. Daher stellt sich die Frage: **Wie berechnet man den Gradienten $\nabla_\theta \mathcal{L}$ für ein Beispielpaar (x, y)?** Die letzte Zeile in (4.5) zeigt $\hat{y}$ bei fix vorgegebenem x explizit als Funktion der θ's. Auf den ersten Blick scheint das recht einfach zu gehen – wir müssen ja lediglich die Kettenregel anwenden. Das Problem dabei ist die immens große Zahl an Parametern in realistischen Anwendungen. Wir müssen, in Komponenten angeschrieben, die partiellen Ableitungen

$$\frac{\partial \mathcal{L}}{W^{(1)}_{rs}}, \quad \frac{\partial \mathcal{L}}{b^{(1)}_{r}}, \quad \frac{\partial \mathcal{L}}{W^{(2)}_{rs}}, \quad \frac{\partial \mathcal{L}}{b^{(2)}_{r}}, \quad \frac{\partial \mathcal{L}}{W^{(3)}_{rs}} \quad \text{und} \quad \frac{\partial \mathcal{L}}{b^{(3)}_{r}} \tag{6.2}$$

an einem gegebenen Punkt $\theta \equiv (W^{(1)}, W^{(2)}, W^{(3)}, b^{(1)}, b^{(2)}, b^{(3)})$ berechnen, und das sind *sehr viele* Zahlen! Ist etwa $W^{(1)}$ eine 1000×1000-Matrix, so stellen ihre Komponenten $W^{(1)}_{rs}$ eine Million Variablen dar. Da möchte man bei der Anwendung der Kettenregel möglichst ökonomisch vorgehen. Rechnet man die partiellen Ableitungen in der Reihenfolge, in der sie in (6.2) aufgelistet sind, stur mit der Kettenregel aus, so müssen zahlreiche Größen mehrfach berechnet werden, und damit handelt man sich sehr lange Rechenzeiten ein, viel längere als eigentlich nötig! Und damit sind wir beim entscheidenden Punkt angelangt. Die Lösung des Problems besteht darin, die Reihenfolge der Berechnungen umzukehren! Wie und warum das funktioniert, das wollen wir uns anhand unseres Feedforward-Netzes mit zwei verdeckten Schickten nun noch genauer ansehen.

7 Berechnung des Gradienten der Kostenfunktion

Zunächst besagt die Kettenregel

$$\nabla_\theta \mathcal{L}(\theta) = \sum_k \frac{\partial L}{\partial \hat{y}_k}(\hat{y}(x,\theta), y)\, \nabla_\theta \hat{y}_k(x,\theta). \tag{7.1}$$

Wir schreiben sie der Einfachheit in der kürzeren Form

$$\nabla_\theta \mathcal{L} = \sum_k \frac{\partial L}{\partial \hat{y}_k}\, \nabla_\theta \hat{y}_k \tag{7.2}$$

an, müssen dabei aber die Argumente, die in (7.1) angeschrieben sind, stets mitdenken. So nimmt (7.2) beispielsweise für die Variablen $W^{(1)}_{rs}$, die ja Bestandteile von θ sind, die Form

$$\frac{\partial \mathcal{L}}{\partial W^{(1)}_{rs}} = \sum_k \frac{\partial L}{\partial \hat{y}_k}\, \frac{\partial \hat{y}_k}{\partial W^{(1)}_{rs}} \tag{7.3}$$

an, für die Variablen $b^{(3)}_r$ die Form

$$\frac{\partial \mathcal{L}}{\partial b^{(3)}_{r}} = \sum_k \frac{\partial L}{\partial \hat{y}_k}\, \frac{\partial \hat{y}_k}{\partial b^{(3)}_{r}}, \tag{7.4}$$

und analoge Formeln erhalten wir für $W^{(2)}_{rs}$, $W^{(3)}_{rs}$, $b^{(1)}_r$ und $b^{(2)}_r$. Der erste Faktor in der Summe (7.2) ist in der Regel leicht berechnet. Für die Kostenfunktion (5.1) erhalten wir

$$\frac{\partial L}{\partial \hat{y}_k} = \hat{y}_k - y_k \,. \tag{7.5}$$

Interessant wird es bei der Berechnung des zweiten Faktors, also des θ-Gradienten von $\hat{y}$.

Wie $\hat{y}$ explizit von den Parametern abhängt, sehen wir am besten in der letzten Zeile von (4.5), aber um etwa den Gradienten nach den Komponenten von $b^{(3)}$ zu berechnen, genügt der letzte Ausdruck in der *ersten* Zeile von (4.5). Jetzt machen sich die zuvor eingeführten Abkürzungen für die Zwischenergebnisse und die vielen Schreibweisen in (4.5) bezahlt. Wir beginnen also mit jenen Komponenten von $\nabla_\theta \hat{y}_k$, die den partiellen Ableitungen nach den Komponenten von $b^{(3)}$ entsprechen und wollen berechnen:

$$\frac{\partial \hat{y}_k}{\partial b^{(3)}_r} = ? \tag{7.6}$$

Nun sind die in (4.5) eingeführten Abkürzungen hilfreich:

$$\frac{\partial \hat{y}_k}{\partial b^{(3)}_r} = \frac{\partial}{\partial b^{(3)}_r} f\big(\underbrace{(W^{(3)}u^{(2)})_k + b^{(3)}_k}_{v^{(3)}_k} \big) = \left\{ \begin{array}{ll} 0 & \text{wenn } r \neq k \\ f'(v^{(3)}_k) & \text{wenn } r = k \end{array} \right\} = \delta_{kr}\, f'(v^{(3)}_k). \tag{7.7}$$

Damit ergeben sich die Komponenten des Gradienten von $\mathcal{L}$ bezüglich der Variable $b^{(3)}_r$ gemäß (7.4) zu

$$\frac{\partial \mathcal{L}}{\partial b^{(3)}_r} = \sum_k \frac{\partial L}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial b^{(3)}_r} = \sum_k \frac{\partial L}{\partial \hat{y}_k}\, \delta_{kr}\, f'(v^{(3)}_k) = \frac{\partial L}{\partial \hat{y}_r}\, f'(v^{(3)}_r) \,. \tag{7.8}$$

Da die Aktivierungsfunktion f in der Regel durch einen sehr einfachen Term gegeben ist (einige Beispiele haben wir ja bereits angegeben), besitzt auch deren Ableitung f' einen einfachen Term, der leicht an einer gegebenen Stelle ausgewertet werden kann. Hier die Ableitungen der oben angegebenen Beispiele für Aktivierungsfunktionen:

- Die Ableitung der ReLU-Funktion (3.1) an einer Stelle $x < 0$ ist $f'(x) = 0$, ihre Ableitung an einer Stelle $x > 0$ ist $f'(x) = 1$. An der Stelle $x = 0$ ist die ReLU-Funktion nicht differenzierbar, besitzt also hier keine Ableitung, aber die Erfahrung hat gezeigt, dass dieser Mangel in der Praxis keine Probleme macht, da das Argument, auf das f' angewandt werden muss, in Berechnungen mit neuronalen Netzen – wie beispielsweise in (7.8) – kaum jemals exakt gleich 0 ist. Welchen Wert man für $f'(0)$ veranschlagt, sollte er doch einmal auftreten, ist Geschmackssache. Es bieten sich 0, $\frac{1}{2}$ und 1 an. Man kann für f' also die bekannte Heavisidesche Stufenfunktion (*unit step function*) H setzen:

$$H(x) = \left\{ \begin{array}{ll} 0 & \text{wenn } x < 0 \\ 1 & \text{wenn } x \geq 0. \end{array} \right. \tag{7.9}$$

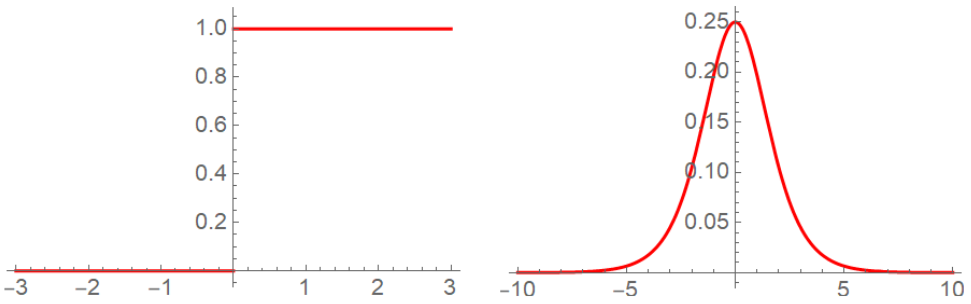


Abbildung 2: Links der Graph der Ableitung (7.9) der ReLU-Funktion (3.1) – dass die Ableitung der ReLU-Funktion an der Stelle 0 nicht existiert, stört in der Praxis nicht –, rechts der Graph der Ableitung (7.10) der Sigmoid-Funktion (3.2).

- Die Ableitung der Sigmoid-Funktion (3.2) ist gegeben durch

$$f'(x) = \frac{e^{-x}}{(1+e^{-x})^2} = f(x)\big(1-f(x)\big). \tag{7.10}$$

  Besonders die letzte Form ist verlockend einfach, weil sie beim Programmieren nur zwei Aufrufe der ohnehin zur Verfügung stehenden Funktion f benötigt.

In allen Fällen ist also f' an einer gegebenen Stelle leicht auszuwerten. Kehren wir zu (7.8) zurück. Dort ist f' an der Stelle $v_r^{(3)}$ auszuwerten. Aber wie bekommen wir $v_k^{(3)}$? Und jetzt denken wir ein bisschen „informatisch“: Der springende Punkt ist, dass man $\hat{y}$ ja (im Ernstfall natürlich mit dem Computer) bereits berechnet und damit auch alle Zwischenschritte in (4.5) durchlaufen hat. Mit anderen Worten: Die in (7.8) benötigte Stelle $v_k^{(3)}$, also die k-te Komponente von $v^{(3)}$, an der f' ausgewertet werden muss, wurde bereits ermittelt und steht für weitere Berechnungen zur Verfügung! Auf diese Weise können also die Komponenten des Gradienten von $\mathcal{L}$ bezüglich $b_r^{(3)}$ leicht berechnet werden. Beachten Sie, dass wir, wie angekündigt, mit der partiellen Ableitung nach dem letzten Parameter in der Liste (6.2) begonnen haben. Hätten wir gleich zu Beginn nach $b_r^{(1)}$ oder nach $W_{rs}^{(1)}$ partiell ableiten wollen, hätte man sich mit Hilfe der Kettenregel durch alle Funktionen, die das Netz anwendet, vorkämpfen müssen. Wir werden gleich sehen, dass sich die partiellen Ableitungen nach den „früheren“ Parametern auf jene nach den „späteren“ zurückführen lassen. Die Gradientenberechnung in einem Feedforward-Netz wird also gewissermaßen in umgekehrter Reihenfolge durchgeführt (weshalb sie auch als **Backpropagation** bezeichnet wird).

Die nächste partielle Ableitung von $\hat{y}_k$, die wir berechnen, ist

$$\frac{\partial \hat{y}_k}{\partial W_{rs}^{(3)}} = ? \tag{7.11}$$

Gemäß der Kettenregel (die wir auch im Folgenden oft anwenden, ohne das jedesmal eigens dazuzusagen) treten zunächst Terme mit den Komponenten von $f'(v^{(3)})$ auf, die wir aber

gerade ermittelt haben! Dazu kommt noch die innere Ableitung, so dass wir insgesamt finden

$$\frac{\partial \hat{y}_k}{\partial W^{(3)}_{rs}} = \frac{\partial}{\partial W^{(3)}_{rs}} f\big(\underbrace{\sum_l W^{(3)}_{kl} u^{(2)}_l + b^{(3)}_k}_{v^{(3)}_k}\big) = f'(v^{(3)}_k)\,\frac{\partial}{\partial W^{(3)}_{rs}} \sum_l W^{(3)}_{kl} u^{(2)}_l =$$

$$= \delta_{kr}\, f'(v^{(3)}_k)\, u^{(2)}_s\,. \tag{7.12}$$

Damit ergeben sich die Komponenten des Gradienten von $\mathcal{L}$ bezüglich der Variable $W^{(3)}_{rs}$ zu

$$\frac{\partial \mathcal{L}}{\partial W^{(3)}_{rs}} = \sum_k \frac{\partial L}{\partial \hat{y}_k}\,\frac{\partial \hat{y}_k}{\partial W^{(3)}_{rs}} = \sum_k \frac{\partial L}{\partial \hat{y}_k}\,\delta_{kr}\, f'(v^{(3)}_k)\, u^{(2)}_s = \frac{\partial L}{\partial \hat{y}_r}\, f'(v^{(3)}_r)\, u^{(2)}_s\,. \tag{7.13}$$

Und jetzt schauen Sie nach oben, zu (7.8), und sehen, dass es sich bei $\dfrac{\partial L}{\partial \hat{y}_r}\, f'(v^{(3)}_r)$ genau um die bereits berechneten Komponenten des Gradienten von $\mathcal{L}$ bezüglich $b^{(3)}_r$ handelt! Wir erhalten also

$$\frac{\partial \mathcal{L}}{\partial W^{(3)}_{rs}} = \frac{\partial \mathcal{L}}{\partial b^{(3)}_r}\, u^{(2)}_s\,, \tag{7.14}$$

wobei der erste Faktor zuvor mit Hilfe von (7.8) ermittelt wurde – also bereits ohne weitere Berechnung zur Verfügung steht – und $u^{(2)}_s$ im Zuge der Berechnung von $\hat{y}$ ermittelt wurde und damit ebenfalls bereits zur Verfügung steht. Jede der Komponenten des Gradienten von $\mathcal{L}$ bezüglich der Komponenten von $W^{(3)}$ ist also einfach das Produkt zweier Zahlen, die an diesem Punkt der Berechnung bereits bekannt sind!

Wir fahren in analoger Weise fort, um die verbleibenden partiellen Ableitungen zu berechnen:

$$\begin{aligned}
\frac{\partial \hat{y}_k}{\partial b^{(2)}_r} &= \frac{\partial}{\partial b^{(2)}_r} f\Big(\underbrace{\sum_l W^{(3)}_{kl} f\big((W^{(2)}u^{(1)})_l + b^{(2)}_l\big) + b^{(3)}_k}_{v^{(3)}_k}\Big) = \\
&= f'(v^{(3)}_k)\,\frac{\partial}{\partial b^{(2)}_r}\left(\sum_l W^{(3)}_{kl} f\big((W^{(2)}u^{(1)})_l + b^{(2)}_l\big) + b^{(3)}_k\right) = \\
&= f'(v^{(3)}_k) \sum_l W^{(3)}_{kl}\,\frac{\partial}{\partial b^{(2)}_r} f\big(\underbrace{(W^{(2)}u^{(1)})_l + b^{(2)}_l}_{v^{(2)}_l}\big) = \\
&= f'(v^{(3)}_k) \sum_l W^{(3)}_{kl} f'(v^{(2)}_l)\,\underbrace{\frac{\partial}{\partial b^{(2)}_r} b^{(2)}_l}_{\delta_{rl}} = f'(v^{(3)}_k)\, W^{(3)}_{kr} f'(v^{(2)}_r).
\end{aligned} \tag{7.15}$$

Damit ergeben sich die Komponenten des Gradienten von $\mathcal{L}$ bezüglich der Variable $b^{(2)}_r$ zu

$$\frac{\partial \mathcal{L}}{\partial b^{(2)}_r} = \sum_k \frac{\partial L}{\partial \hat{y}_k}\,\frac{\partial \hat{y}_k}{\partial b^{(2)}_r} = \sum_k \frac{\partial L}{\partial \hat{y}_k}\, f'(v^{(3)}_k)\, W^{(3)}_{kr} f'(v^{(2)}_r)\,. \tag{7.16}$$

Und jetzt schauen wir wieder nach oben, zu (7.8), und sehen, dass es sich bei $\frac{\partial L}{\partial \hat{y}_k} f'(v_k^{(3)})$ um die bereits berechneten Komponenten des Gradienten von $\mathcal{L}$ bezüglich $b_k^{(3)}$ handelt. Wir erhalten also

$$\frac{\partial \mathcal{L}}{\partial b_r^{(2)}} = \sum_k \frac{\partial \mathcal{L}}{\partial b_k^{(3)}} W_{kr}^{(3)} f'(v_r^{(2)}). \tag{7.17}$$

Wieder sehen wir, dass alle Bestandteile dieses Ausdrucks bereits früher berechnet worden sind. Im nächsten Schritt berechnen wir

$$\frac{\partial \hat{y}_k}{\partial W_{rs}^{(2)}} = ? \tag{7.18}$$

Dazu schreiben wir den bereits in (7.15) auftretenden Ausdruck $(W^{(2)}u^{(1)})_l$ in der Form $\sum_q W_{lq}^{(2)} u_q^{(1)}$ und finden

$$\begin{aligned} \frac{\partial \hat{y}_k}{\partial W_{rs}^{(2)}} &= f'(v_k^{(3)}) \sum_l W_{kl}^{(3)} f'(v_l^{(2)}) \underbrace{\frac{\partial}{\partial W_{rs}^{(2)}} \sum_q W_{lq}^{(2)} u_q^{(1)}}_{\delta_{rl} u_s^{(2)}} \\ &= f'(v_k^{(3)})\, W_{kr}^{(3)} f'(v_r^{(2)})\, u_s^{(1)} \end{aligned} \tag{7.19}$$

und damit

$$\frac{\partial \mathcal{L}}{\partial W_{rs}^{(2)}} = \sum_k \frac{\partial L}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial W_{rs}^{(2)}} = \sum_k \frac{\partial L}{\partial \hat{y}_k} f'(v_k^{(3)})\, W_{kr}^{(3)} f'(v_r^{(2)})\, u_s^{(1)}\,. \tag{7.20}$$

Ein Blick auf (7.16) zeigt, dass das gleich

$$\frac{\partial \mathcal{L}}{\partial W_{rs}^{(2)}} = \frac{\partial \mathcal{L}}{\partial b_r^{(2)}} u_s^{(1)} \tag{7.21}$$

und damit nur ein Produkt aus zwei bereits zuvor berechneten Zahlen ist.

Die letzten Schritte werden ganz analog durchgeführt. Wir überspringen die Details (versuchen Sie selbst, die Berechnungen auszuführen!) und notieren nur die Ergebnisse:

$$\frac{\partial \hat{y}_k}{\partial b_r^{(1)}} = f'(v_k^{(3)}) \sum_l W_{kl}^{(3)}\, f'(v_l^2)\, W_{lr}^{(2)} f'(v_r^{(1)}) \tag{7.22}$$

$$\frac{\partial \mathcal{L}}{\partial b_r^{(1)}} = \sum_k \frac{\partial \mathcal{L}}{\partial b_k^{(2)}} W_{kr}^{(2)} f'(v_r^{(1)}) \tag{7.23}$$

$$\frac{\partial \hat{y}_k}{\partial W_{rs}^{(1)}} = f'(v_k^{(3)}) \sum_l W_{kl}^{(3)}\, f'(v_l^2)\, W_{lr}^{(2)} f'(v_r^{(1)})\, x_s \tag{7.24}$$

$$\frac{\partial \mathcal{L}}{\partial W_{rs}^{(1)}} = \frac{\partial \mathcal{L}}{\partial b_r^{(1)}} x_s\,. \tag{7.25}$$

Damit haben wir die Berechnung des Gradienten von $\mathcal{L}$ nach den Parametern unseres Feedforward-Netzes erfolgreich und auf sparsame Weise durchgeführt. $\nabla_\theta \mathcal{L}$ besteht aus den Komponenten

(7.8), (7.14), (7.17), (7.21), (7.23) und (7.25). Um die rekursive Natur dieser Ergebnisse noch einmal hervorzuheben, schreiben wir diese Beziehungen in der Form

$$\frac{\partial \mathcal{L}}{\partial b_r^{(3)}} = \frac{\partial L}{\partial \hat{y}_r} f'(v_r^{(3)}) \tag{7.26}$$

$$\frac{\partial \mathcal{L}}{\partial W_{rs}^{(3)}} = \frac{\partial \mathcal{L}}{\partial b_r^{(3)}} u_s^{(2)} \tag{7.27}$$

$$\left.\begin{aligned} \frac{\partial \mathcal{L}}{\partial b_r^{(\alpha)}} &= \sum_k \frac{\partial \mathcal{L}}{\partial b_k^{(\alpha+1)}} W_{kr}^{(\alpha+1)} f'(v_r^{(\alpha)}) \\ \frac{\partial \mathcal{L}}{\partial W_{rs}^{(\alpha)}} &= \frac{\partial \mathcal{L}}{\partial b_r^{(\alpha)}} u_s^{(\alpha-1)} \end{aligned}\right\} \quad \alpha = 2, 1 \tag{7.28}$$

an. Beachten Sie, dass hier α von 2 nach 1 zurückgezählt wird und dass, wenn man bei $\alpha = 1$ angelangt ist, $u_s^{(0)} = x_s$ gesetzt wird. Damit sollte klar sein, wie die entsprechenden Formeln für ein Feedforward-Netz mit K verdeckten Schichten aussehen: Die ersten beiden Berechnungen werden mit $(K+1)$ statt (3) und (K) statt (2) ausgeführt, und danach wird α von K bis 1 zurückgezählt.

Ohne näher darauf einzugehen sei erwähnt, dass auch höhere Ableitungen von $\mathcal{L}$ nach den Parametern (die benötigt werden, wenn die Optimierung mit dem Gradientenabstiegsverfahren zweiter Ordnung durchgeführt werden soll) in ähnlicher Weise berechnet werden können.

8 Funktionsapproximation

Ab den späten 1980er-Jahren wurde in einer Reihe von Theoremen gezeigt, dass eine sehr große Klasse von Funktionen $\mathbb{R}^n \to \mathbb{R}^m$, darunter praktisch alle, die in Problemen der künstischen Intelligenz auftreten, beliebig genau durch ein geeignetes Feedforward-Netz mit linearer Ausgabeschicht approximiert werden können, wobei die Aktivierungsfunktion weitgehend beliebig ist. Man nennt dies die **universelle Approximationseigenschaft** neuronaler Netze. Tatsächlich kann man sich bei dieser Aussage sogar auf mit Feedforward-Netze mit einzige verdeckte Schicht beschränken! Neuronale Netze sind also *im Prinzip* sehr mächtig.

Wir verzichten auf die Details (z.B. wie die Güte einer solchen Approximation gemessen wird), wollen uns aber kurz einen ganz groben Eindruck verschaffen, wie groß die Menge der auf diese Weise approximierbaren Funktionen ist. Beschränken wir uns dazu auf den Fall $m = n = 1$ (d.h. auf Funktionen $\mathbb{R} \to \mathbb{R}$) und legen als Aktivierungsfunktion die ReLU-Funktion (3.1) zugrunde. Welche Funktionen können auf diese Weise *exakt* dargestellt werden? Es sind dies alle Funktionen, deren Graph aus endlich vielen Geradenstücken besteht, die lückenlos aneinandergereiht werden: Durch die Anwendung einer affinen Abbildung und der ReLU-Funktion auf die Eingabe x bekommt man bekommt man Funktionen, deren Graph aus zwei Geradenstücken besteht. In einem Feedforward-Netz können solche Funktionen nach Art eines Parallelrechners beliebig linearkombiniert werden. Tatsächlich kann jede Funktion, deren Graph aus endlich vielen lückenlos aneinandergereihten Geradenstücken besteht, durch ein geeignetes Feedforward-Netz mit einer einzigen verdeckten Schicht exakt dargestellt werden. In der Ausgabeschicht wird auf

die Anwendung der Aktivierungsfunktion verzichtet, damit der berechnete Wert nicht auf den Bildbereich dieser Funktion (also im Fall der ReLU-Funktion die Menge der nicht negativen reellen Zahlen) beschränkt ist (daher der Forderung nach einer linearen Ausgabeschicht in der obigen Formulierung).

Sie können sich selbst leicht klar machen, wie das im Detail funktioniert: Jede auf einem vorgegebenen beschränkten Intervall $[a, b]$ definierte stetige reelle Funktion g, die in N Teilintervallen $[a_j, a_{j+1}]$ für $j = 0, 1, \ldots, N-1$ mit $a = a_0 < a_1 < \cdots < a_N = b$ affin ist (deren Graph daher aus den entsprechenden Geradenstücken besteht), kann in der Form

$$g(x) = \sum_{j=0}^{N-1} \lambda_j f(x - a_j) + c \tag{8.1}$$

mit Koeffizienten $\lambda_j, c \in \mathbb{R}$ angesetzt werden. Sind die Funktionswerte $g(a_0) = g_0$, $g(a_1) = g_1, \ldots, g(a_N) = g_N$ beliebig vorgegeben, so sind alle Koeffizienten eindeutig bestimmt und lassen sich leicht in rekursiver Weise ermitteln:

$$\begin{aligned}
&\text{Löse die Gleichung } g(a_0) = g_0 \text{ nach } c \\
&\text{Löse die Gleichung } g(a_1) = g_1 \text{ nach } \lambda_0 \\
&\text{Löse die Gleichung } g(a_2) = g_2 \text{ nach } \lambda_1 \\
&\ldots\ldots\ldots\ldots \\
&\text{Löse die Gleichung } g(a_N) = g_N \text{ nach } \lambda_{N-1}
\end{aligned} \tag{8.2}$$

(Schreiben Sie zur Übung die Lösungen der Gleichungen an! Überlegen Sie, wie die Berechnung von g durch ein Feedforward-Netz mit einer einzigen verdeckten Schicht und einer linearen Ausgabeschicht dargestellt werden kann!) Ist nun eine beliebige stetige Funktion h auf einem Intervall $[a, b]$ vorgegeben, so wählt man Zwischenpunkte $a = a_0 < a_1 < \cdots < a_N = b$ und approximiert h durch eine stetige, stückweise affine Funktion vom Typ (8.1). Intuitiv ist klar, dass die Approximation beliebig gut gemacht werden kann, indem die Zwischenpunkte (bei entsprechend großem N) beliebig nahe aneinandergerückt werden. In höheren Dimensionen, also wenn es um Funktionen $\mathbb{R}^n \to \mathbb{R}^m$ geht, wird die Sachlage zwar komplizierter, die Hauptaussage bleibt aber im Prinzip gleich: Neuronale Netze sind in der Lage, eine riesige Anzahl von Funktionen beliebig genau zu approximieren.

Die Freude über die universelle Approximationseigenschaft der neuronalen Netze wird aber durch zwei Umstände getrübt:

- Dass ein neuronales Netz mit nur einer einzigen verdeckten Schicht für eine beliebig gute Approximation ausreicht, bedeutet nicht, dass das die sparsamste Möglichkeit ist. Wieviele verdeckte Schichten am günstigsten für die Approximation einer gegebenen Funktion sind, und wieviele Neuronen sie enthalten, ist damit nicht gesagt.
- Selbst wenn die Anzahl der Schichten und die Anzahl der Neuronen der einzelnen Schichten einmal gewählt sind, gibt es keine Garantie, dass das Gradientenabstiegsverfahren die günstigste Wahl der Parameter θ liefert.

Daher ist man in der Praxis bei der Planung neuronaler Netz in hohem Maße auf Erfahrungswerte uud Versuche angewiesen. Gleichzeitig etabliert sich derzeit ein mathematisches Forschungsgebiet, dessen Ziel es ist, neuronale Netze in all ihren Varianten (von denen die Feedforward-Netze nur ein Spezialfall sind) besser zu verstehen und ihre Nutzung für immer komplexere Probleme möglichst effizient zu gestalten.

Dieses Skriptum wurde erstellt im Oktober 2020 für den Master-Studiengang „Data Science“ an der Fachhochschule Technikum Wien (http://www.technikum-wien.at/).
Die Skripten-Seite von mathe online (http://www.mathe-online.at/) finden Sie unter http://www.mathe-online.at/skripten/.