Theorie des maschinellen Lernens

Hans U. Simon

21. Juli 2017

20 Neuronale Netzwerke

Künstliche neuronale Netzwerke sind Berechnungsmodelle, deren Design auf einer groben Modellierung natürlicher neuronaler Netzwerke (wie wir sie zum Beispiel im Gehirn vorfinden) basiert. Die Knoten dieser Netzwerke repräsentieren die Nervenzellen, welche auch Neurone genannt werden. Die Kanten repräsentieren die Vernetzungsstruktur. Mathematisch gesehen bilden die Neurone und ihre Verbindungen einen gerichteten Graphen. Ist dieser Graph azyklisch, so spricht man von einem *vorwärtsgerichteten neuronalen Netzwerk*, das auch „Feedforward Neural Network“ oder kurz „FFNN“ genannt wird.

Die Kanten e in einem neuronalen Netzwerk sind mit einem reellen Parameter $w(e)$ gewichtet. Der Absolutbetrag $|w(e)|$ steht für die Stärke der Reizübertragung; mit dem Vorzeichen unterscheiden wir exzitatorische (anregende) von inhibitorischer (hemmender) Reizübertragung. Die Gewichtsparameter sind programmierbar. Durch die Festlegung der Gewichte kann eine Anpassung des Netzwerkes an Trainingsdaten erreicht werden.

In diesem Kapitel werden wir uns auf die Diskussion vorwärtsgerichteter neuronaler Netzwerke konzentrieren. In Abschnitt 20.1 führen wir ein FFNN-Standardmodell ein, welches in Abschnitt 20.2 geringfügig erweitert wird. Abschnitt 20.3 beschäftigt sich mit der Rechenkraft von FFNN. Die Informationskomplexität einer FFNN-Architektur und der durch sie repräsentierten Hypothesenklasse wird in Abschnitt 20.4 analysiert. Im abschließenden Abschnitt 20.5 besprechen wir die Heuristik namens „Backpropagation“, die häufig zur Adjustierung der Gewichtsparameter verwendet wird.

20.1 Das FFNN-Standardmodell

Ein *vorwärtsgerichtetes neuronales Netzwerk $\mathcal{N}$ (auch „Feedforward Neural Network“ oder „FFNN“ genannt)* wird durch folgende Komponenten spezifiziert:

- ein gerichteter azyklischer Graph (V, E) mit Kantengewichten $w : E \to \mathbb{R}$, dessen Knoten als *Neurone* bezeichnet werden
- eine Funktion $\sigma : \mathbb{R} \to \mathbb{R}$, genannt *Aktivierungsfunktion*

Diese Spezifikation wird in der Gleichung $\mathcal{N} = (V, E, \sigma, w)$ zusammengefasst. Das Tripel (V, E, σ) heißt die *Architektur* des Netzwerkes. Die Gewichte $w(e)$ mit $e \in E$ sind programmierbare Parameter, deren Belegung mit reellen Zahlen durch einen Lernalgorithmus fixiert werden kann. Durch die konkrete Wahl der Gewichte passt sich das Netzwerk an vorliegende Trainingsdaten an.

Wir setzen im Folgenden voraus, dass das Netzwerk $\mathcal{N}$ *geschichtet (layered)* ist, d.h., die Knotenmenge V besitzt eine Zerlegung der Form $V = V_0 \cup V_1 \cup \ldots \cup V_T$, und die Kanten in E verlaufen stets von einer Schicht zu der darauf folgenden, d.h., $E \subseteq \cup_{t=1}^{T}(V_{t-1} \times V_t)$. Wir nennen V_0 die *Eingabeschicht (input layer)*, $V_1, \ldots, V_{T-1}$ die *verborgenen Schichten (hidden layers)* und V_T die *Ausgabeschicht (output layer)*. T heißt die *Tiefe* und $\max_{1 \leq t \leq T} |V_t|$ heißt die *Weite* von $\mathcal{N}$.

Es sei $n = |V_0| - 1$ und $d = |V_T|$. Dann repräsentiert $\mathcal{N} = (V, E, \sigma, w)$ eine Funktion $h_{\mathcal{N}} : \mathbb{R}^n \to \mathbb{R}^d$. Es folgt eine Beschreibung, wie die Berechnung auf einer Netzeingabe $x \in \mathbb{R}^n$ die Schichten $0, 1, \ldots, T$ der Reihe nach durchläuft, wobei $o_v(x)$ den Wert bezeichnet, den ein Knoten $v \in V$ berechnet:

1. Die Eingabeschicht V_0 reicht im Wesentlichen x an die erste Schicht V_1 weiter. Genauer: für $i = 1, \ldots, n$ produziert der i-te Knoten in V_0 die Ausgabe x_i (= die i-te Komponente der Netzeingabe x) und der $(n+1)$-te Knoten in V_0 produziert die Ausgabe 1.

2. Nehmen wir induktiv an, dass die Neurone der t-ten Schicht ihre Ausgabewerte schon berechnet haben und betrachten nun ein Neuron v in V_{t+1}. Es seien $u_1, \ldots, u_r$ alle Neurone in V_t, die durch eine Kante mit v verbunden sind. Weiter seien $w_1, \ldots, w_r$ die diesen Kanten zugeordneten Gewichte. Wir bezeichnen dann

$$a_v(x) := \sum_{i=1}^{r} w_i o_{u_i}(x) \tag{1}$$

   als die *Eingabe für den Knoten* v und dieser berechnet

$$o_v(x) := \sigma(a_v(x)) = \sigma\left(\sum_{i=1}^{r} w_i o_{u_i}(x)\right) \quad .$$

Seien $v_1, \ldots, v_d$ die d Knoten der Ausgabeschicht V_T. Dann setzen wir

$$h_{\mathcal{N}}(x) := (o_{v_1}(x), \ldots, o_{v_d}(x)) \in \mathbb{R}^d \quad ,$$

d.h., die von $v_1, \ldots, v_d \in V_T$ berechneten Werte bilden die Ausgabe des Netzwerkes.

Da $\mathcal{N} = (V, E, \sigma, w)$ mit $n + 1 = |V_0|$ und $d = |V_T|$ eine Funktion $h_{\mathcal{N}} : \mathbb{R}^n \to \mathbb{R}^d$ repräsentiert, repräsentiert die Architektur (V, E, σ) die Hypothesenklasse

$$\mathcal{H}_{V,E,\sigma} := \{h_{V,E,\sigma,w} |\ w : E \to \mathbb{R}\} \subseteq \{f : \mathbb{R}^n \to \mathbb{R}^d\} \quad .$$

Das Adjustieren der Gewichtsparameter w an vorliegende Trainingsdaten entspricht also der Wahl einer Hypothese aus der Klasse $\mathcal{H}_{V,E,\sigma}$.

20.2 Erweiterung des Standardmodells

Wir erweitern $\mathcal{N}$ um eine Komponente $\theta : V \to \mathbb{R}$, so dass $\mathcal{N} = (V, E, \sigma, w, \theta)$. Die Eingabe $a_v(x)$ für einen Knoten v bei Netzeingabe x — vergleiche mit (1) — hat dann die Form

$$a_v(x) := \sum_{i=1}^{r} w_i o_{u_i}(x) + \theta(v) \ ,$$

d.h., θ assoziiert zu jedem Knoten ein Bias. Ansonsten verläuft die Berechnung der Funktion $h_{\mathcal{N}}$ genau so wie wir es in Abschnitt 20.1 beschrieben haben. Wir betrachten die Biaswerte θ ebenso wie die Gewichte w als programmierbare Parameter. Somit repräsentiert eine Architektur (V, E, σ) die Hypothesenklasse

$$\mathcal{H}_{V,E,\sigma} := \{h_{V,E,\sigma,w,\theta} |\ w : E \to \mathbb{R}, \theta : V \to \mathbb{R}\} \subseteq \{f : \mathbb{R}^n \to \mathbb{R}^d\} \ .$$

Es ist nicht schwer zu zeigen, dass jedes neuronale Netzwerk in dem erweiterten Modell simulierbar ist durch ein (nicht wesentlich größeres) neuronales Netzwerk im Standardmodell. Hierzu muss lediglich die 1-Konstante (welche in der Eingabeschicht zur Verfügung gestellt wird) mit Hilfe eines Extra-Neurons pro versteckter Schicht weiterpropagiert werden, so dass ein Bias $\theta(v)$ mit Hilfe eines zusätzlichen Gewichtsparameters einer in v mündenden Kante dargestellt werden kann. Die Details sind leicht einzufüllen. Im Folgenden bedienen wir uns meistens des erweiterten Modells.

20.3 Die Rechenkraft von FFNN

Für eine Boolesche Variable v_i identifizieren wir v_i^1 mit v_i und v_i^0 mit ihrem Negat $\overline{v_i}$. Es sei $b = (b_1, \ldots, b_n) \in \{0,1\}$ ein Boolescher Vektor. Dann berechnet der Term $T = v_1^{b_1} \wedge \ldots \wedge v_n^{b_n}$, angewendet auf einen weiteren Booleschen Vektor $a = (a_1, \ldots, a_n)$ den Wert $T(a) = 1$ genau dann, wenn $a = b$. Es sei $f : \{0,1\}^n \to \{0,1\}$ eine beliebige Boolesche Funktion in n Variablen. Weiter sei $B(f) = f^{-1}(1)$ die Menge der Booleschen Vektoren, die von f auf 1 abgebildet werden. Offensichtlich kann f durch die folgende DNF-Formel dargestellt werden:

$$f(v) = \bigvee_{b \in B(f)} (v_1^{b_1} \wedge \ldots \wedge v_n^{b_n}) \ .$$

Es ergibt sich folgendes Resultat:

Lemma 20.1 *Jede Boolesche Funktion kann durch eine DNF-Formel dargestellt werden.*

Es bezeichne $\oplus$ die Addition modulo 2 und es sei $P(v_1, \ldots, v_n) = v_1 \oplus \ldots \oplus v_n$ die Funktion, die den Wert 1 genau dann liefert, wenn sich in $v_1, \ldots, v_n$ ungeradzahlig viele Einsen befinden (die sogenannte Paritätsfunktion). Man kann leicht zeigen, dass jede DNF-Formel, welche P darstellt, 2^{n-1} Konjunktionen verwenden muss. Hieraus folgt, dass wir i.A. DNF-Formeln exponentieller Größe benötigen, um alle Booleschen Funktionen in n Variablen darzustellen.

Es seien $v, u_1, \ldots, u_r \in \{v_1, \ldots, v_n\}$ Boolesche Variable. Wir setzen

$$L_b(v) := \begin{cases} 1 - v & \text{falls } b = 0 \\ v & \text{falls } b = 1 \end{cases} .$$

Da die Boolesche Formel v^b den selben Wert (0 oder 1) liefert wie die arithmetische Formel $L_b(v)$, wird $L_b(v)$ auch als „Arithmetisierung“ des Literals v^b bezeichnet. Eine Konjunktion oder Disjunktion von Literalen kann von einem Neuron mit $\text{sign}(\cdot)$ als Aktivierungsfunktion nach folgendem Schema simuliert werden:

$$\begin{aligned} u_1^{b_1} \wedge \ldots \wedge u_r^{b_r} = 1 &\Leftrightarrow L_{b_1}(u_1) + \ldots + L_{b_n}(u_r) \geq r \Leftrightarrow L_{b_1}(u_1) + \ldots + L_{b_n}(u_r) - r \geq 0 \\ u_1^{b_1} \vee \ldots \vee u_r^{b_r} = 1 &\Leftrightarrow L_{b_1}(u_1) + \ldots + L_{b_r}(u_r) \geq 1 \Leftrightarrow L_{b_1}(u_1) + \ldots + L_{b_r}(u_r) - 1 \geq 0 \end{aligned}$$

Insbesondere gilt

$$u_1 \vee \ldots \vee u_r = 1 \Leftrightarrow u_1 + \ldots + u_r \geq 1 \Leftrightarrow u_1 + \ldots + u_r - 1 \geq 0 \ .$$

Wir können daher aus Lemma 20.1 direkt folgendes Resultat ableiten:

Lemma 20.2 *Jede DNF-Formel (und damit jede Boolesche Funktion) kann von einem neuronalen Netzwerk der Tiefe 2 berechnet werden.*

Beispiel 20.3 *Wir bauen ein Netzwerk für die DNF-Formel* $(v_1 \wedge \overline{v_3}) \vee (\overline{v_1} \wedge \overline{v_4})$. *Dazu nutzen wir aus:*

$$\begin{aligned} v_1 \wedge \overline{v_3} = 1 &\Leftrightarrow v_1 + (1 - v_3) \geq 2 \Leftrightarrow v_1 - v_3 - 1 \geq 0 \\ \overline{v_1} \wedge \overline{v_4} = 1 &\Leftrightarrow (1 - v_1) + (1 - v_4) \geq 2 \Leftrightarrow -v_1 - v_4 \geq 0 \end{aligned}$$

Dies führt zu dem in Abbildung 1 dargestellten Netzwerk.

Wir werden später zeigen, dass die VC-Dimension von $\mathcal{H}_{V,E,\text{sign}}$ durch $O(|E| \ln(|E|)) = O(|V|^3)$ nach oben beschränkt ist. Da 2^n die VC-Dimension von $\{f : \{0,1\}^n \to \{0,1\}\}$ ist, muss es Boolesche Funktionen geben, die nur von einer Architektur mit $\Omega(2^{n/3})$ Neuronen berechnet werden können. Man benötigt also i.A. Netzwerke einer exponentiellen Größe. Ähnliche Resultate gelten, wenn σ die Sigmoid-Funktion ist.

Neuronale Netzwerkfamilien polynomiell beschränkter Größe. *P/poly* bezeichnet die Klasse aller Familien $(f_n)_{n\geq 1}$ Boolescher Funktionen, die von einer Schaltkreisfamilie $(C_n)_{n\geq 1}$ polynomiell beschränkter Größe über der Standardbasis $\{\neg, \wedge, \vee\}$ berechnet werden können. Da wir solche Schaltkreise (wie weiter oben bereits ausgeführt) mit Hilfe neuronaler Netzwerke effizient simulieren können, kann jedes $(f_n)_{n\geq 1} \in P/poly$ auch durch eine Familie $(\mathcal{N}_n)_{n\geq 1}$ neuronaler Netzwerke einer polynomiell beschränkten Größe berechnet werden. Die Klasse *P/poly* ist recht groß. Sie umfasst zum Beispiel die Klasse P der in Polynomialzeit Turing-berechenbaren Booleschen Funktionsfamilien.

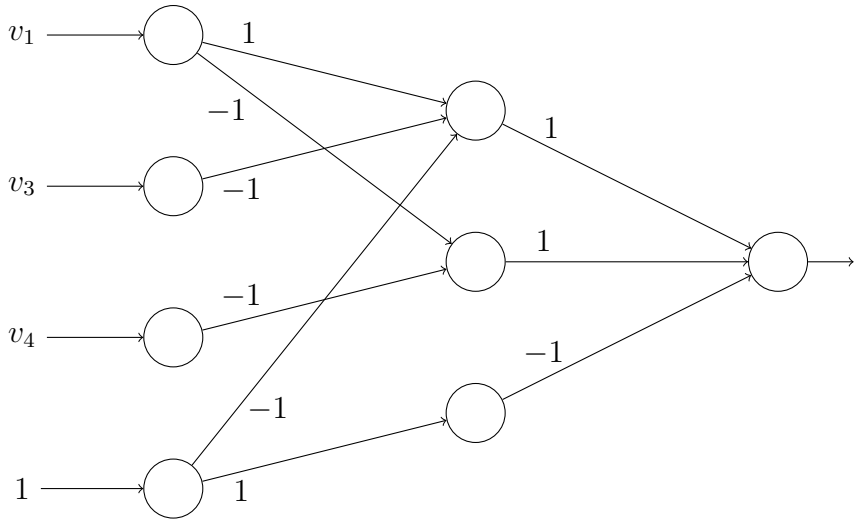
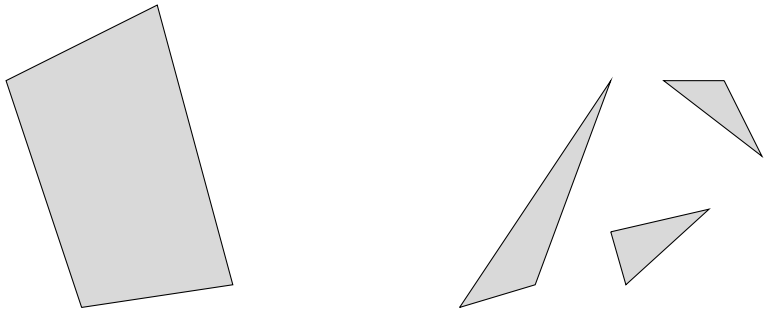


Abbildung 1: Netzwerk für die DNF-Formel $(v_1 \wedge \overline{v_3}) \vee (\overline{v_1} \wedge \overline{v_4})$.



a) b)

Abbildung 2: a) Konvexer Polyeder als Durchschnitt von vier affinen Halbebenen. b) Vereinigung von drei konvexen Polyedern

FFNN und Vereinigung von konvexen Polyedern. Ein Neuron v in der ersten versteckten Schicht berechnet eine Funktion der Form

$$o_v(x) = \text{sign}\left(\sum_{i=1}^{n} w_i x_i + \theta\right) \quad .$$

Das ist die Indikatorfunktion für einen affinen Halbraum in $\mathbb{R}^n$. Ein *konvexer Polyeder* ist ein Durchschnitt von affinen Halbräumen. Die Indikatorfunktion eines Polyeder ist daher die Konjunktion (Ver-Und-ung) von Indikatorfunktionen von affinen Halbräumen. Somit können neuronale Netzwerke mit einer versteckten Schicht die Indikatorfunktionen zu Polyedern berechnen. Mit zwei versteckten Schichten sind dann Indikatorfunktionen von Vereinigungen konvexer Polyeder berechenbar. Abbildung 2 veranschaulicht diese Beobachtungen in $\mathbb{R}^2$.

20.4 Die Informationskomplexität von FFNN

Es sei $\mathcal{H} \subseteq \{h : \mathcal{X} \to \mathcal{Y}\}$ für einen Grundbereich $\mathcal{X}$ und einen endlichen Zielbereich $\mathcal{Y}$. Wie gewohnt bezeichnet $\mathcal{H}_M$ mit $M \subseteq \mathcal{X}$ die Menge der auf M eingeschränkten Funktionen aus $\mathcal{H}$. Der Beweis des folgenden Resultates verwendet die einer binären Hypothesenklasse $\mathcal{H}$ zugeordnete Kapazitätsfunktion $\tau_{\mathcal{H}}(m)$, die wir im Kapitel „Lernen via uniforme Konvergenz“ kennengelernt hatten.

Theorem 20.4 *Es sei (V, E, σ) eine FFNN-Architektur mit einem Ausgabeneuron. Dann gilt:*

$$\mathrm{VCdim}(\mathcal{H}_{V,E,\mathrm{sign}}) = O(|E| \ln(|E|)) \ . \tag{2}$$

Beweis Der Kürze halber setzen wir $\mathcal{H} := \mathcal{H}_{V,E,\mathrm{sign}}$. Es sei $v_1, \ldots, v_N$ eine topologische Sortierung der Neurone in (V, E), so dass aus $(v_i, v_j) \in E$ stets $i < j$ folgt. Insbesondere ist v_N das Ausgabeneuron. Es sei n die Dimension der Netzeingabevektoren. Für $i = 1, \ldots, N$ sei r_i die Anzahl der in v_i eingehenden Kanten aus E. Beachte, dass dann $|E| = \sum_{i=1}^{N} r_i$. Zu gegebenem $w : E \to \mathbb{R}$ bezeichne $h_w^{(i)}(x)$ die vom i-ten Neuron v_i in Abhängigkeit von der Netzeingabe $x \in \mathbb{R}^n$ berechnete Funktion. Betrachte die folgende Familie von Funktionen:

$$\mathcal{H}^{\leq i} = \{x \mapsto (h_w^{(1)}(x), \ldots, h_w^{(i)}(x)) |\ w : E \to \mathbb{R}\}$$

Fixiere eine beliebige Menge $M \subset \mathbb{R}^m$ mit $m = |M|$. Es ist leicht zu sehen[1], dass

$$|\mathcal{H}_M^{\leq i}| \leq |\mathcal{H}_M^{\leq i-1}| \cdot \left(\frac{em}{r_i}\right)^{r_i} \leq |\mathcal{H}_M^{\leq i-1}| \cdot (em)^{r_i} \ .$$

Expandieren dieser Rekursion liefert

$$|\mathcal{H}_M^{\leq N}| \leq \prod_{i=1}^{N} (em)^{r_i} = (em)^{|E|} \ .$$

Wegen $|\mathcal{H}_M| \leq |\mathcal{H}_M^{\leq N}|$ können wir nun schlussfolgern, dass $\tau_{\mathcal{H}}(m) \leq (em)^{|E|}$. Für $m = \mathrm{VCdim}(\mathcal{H})$ müsste gelten $2^m \leq (em)^{|E|}$. Geschicktes Auflösen nach m liefert $m = \mathrm{VCdim}(\mathcal{H}) \leq c|E| \ln(|E|)$ für eine geeignete Konstante $c > 0$. **qed.**

20.5 FFNN und Backpropagation

Das Konsistenzproblem für $\mathcal{H}_{V,e,\sigma}$ ist selbst für einfache Architekturen NP-hart und das Lernproblem für $\mathcal{H}_{V,e,\sigma}$ ist unter bestimmten kryptographischen Voraussetzungen selbst dann hart, wenn der Lernalgorithmus die Form seiner Hypothese frei wählen kann. Daher existieren für FFNN nur heuristische Lernverfahren. Das bekannteste unter ihnen ist „Backpropagation“. Es handelt sich dabei, wie wir im Folgenden herausarbeiten werden, im Wesentlichen um ein stochastisches Gradientenabstiegsverfahren, das eine lokal optimale Wahl der Gewichtsparameter findet.

[1] Das wäre eine gute Übungsaufgabe.

20.5.1 Grundlagen aus der Analysis

Es sei seien $f_1, \ldots, f_m$ differenzierbare Funktionen in n reellen Variablen und $f = (f_1, \ldots, f_m) : \mathbb{R}^n \to \mathbb{R}^m$. Es bezeichne $J_w(f) \in \mathbb{R}^{m \times n}$ die Funktionalmatrix (englisch: Jacobian) von f am Punkt $x = w$, d.h., $J_w(f)[i,j] = \frac{\partial f_i}{\partial x_j}(w)$. Mit anderen Worten: die i-te Zeile von $J_w(f)$ ist der Gradient von f_i am Punkt $x = w$. Betrachte die Komposition $f \circ g : \mathbb{R}^k \to \mathbb{R}^m$ für eine weitere differenzierbare Funktion $g : \mathbb{R}^k \to \mathbb{R}^n$. Gemäß der Kettenregel gilt $J_w(f \circ g) = J_{g(w)}(f) \cdot J_w(g)$.

Beispiel 20.5 *Für $f(w) = Aw$ mit $A \in \mathbb{R}^{m \times k}$ gilt $J_w(f) = A$ für alle $w \in \mathbb{R}^k$.*
Es sei $\sigma : \mathbb{R} \to \mathbb{R}$ die Sigmoid-Funktion, d.h., $\sigma(\theta) = \frac{1}{1+e^{-\theta}}$. Man rechnet leicht nach, dass $\sigma'(\theta) = \frac{1}{(1+e^{\theta})(1+e^{-\theta})}$. Weiter sei $\vec{\sigma} : \mathbb{R}^n \to \mathbb{R}^n$ die Funktion, welche σ komponentenweise anwendet, d.h., für $\vec{\theta} = (\theta_1, \ldots, \theta_n)$ gilt $\vec{\sigma}(\vec{\theta}) = (\sigma(\theta_1), \ldots, \sigma(\theta_n))$. Dann ist $J_{\vec{\theta}}(\vec{\sigma})$ eine $(n \times n)$-Diagonalmatrix mit den Werten $\sigma'(\theta_1), \ldots, \sigma'(\theta_n)$ auf der Hauptdiagonalen. Wir notieren diese Matrix im Folgenden als $\mathrm{diag}(\vec{\sigma'}(\vec{\theta}))$.
Sei nun $A \in \mathbb{R}^{n \times k}$ und $g(w) = Aw$. Dann gilt

$$J_w(\sigma \circ g) = \mathrm{diag}(\vec{\sigma'}(Aw)) \cdot A \ . \tag{3}$$

Die Aktualisierung eines programmierbaren Parameters $w \in \mathbb{R}^k$ im Rahmen eines Gradientenabstieges bezüglich einer Kostenfunktion $f : \mathbb{R}^k \to \mathbb{R}$ hat die allgemeine Form

$$w := w - \eta \nabla f(w) \ ,$$

wobei $\nabla f(w) = J_w(f)$ den Gradienten von f an der Stelle w und $\eta > 0$ die sogenannte „Schrittweite“ oder „Lernrate“ bezeichnet. Der Vektor $-\nabla f(w)$ gibt, vom Punkt w aus gesehen, die Richtung an, in welcher die f-Werte am stärksten abfallen.

20.5.2 Gradientenabstieg und Backpropagation

Wir betrachten eine geschichtete Architektur (V, E, σ), wobei σ die Sigmoidfunktion bezeichnet, $|V_0| = n + 1$ und $|V_t| = k_t$ für $t = 1, \ldots, T$. Jedes Netzwerk $\mathcal{N} = (V, E, \sigma, w)$ berechnet daher eine Funktion $h_w : \mathbb{R}^n \to \mathbb{R}^{k_T}$. Wir verwenden die regularisierte quadratische Verlustfunktion, die einer Hypothese h_w und einem Beispiel $(x, y) \sim \mathcal{D}$ die Kosten

$$\ell(h_w, (x, y)) = \frac{1}{2}\|h_w(x) - y\|^2 + \frac{\lambda}{2}\|w\|^2 \tag{4}$$

zuordnet. Dabei ist, wie üblich, $\lambda > 0$ ein Regularisierungsparameter. Die eigentliche Kostenfunktion ist $L_{\mathcal{D}}(h_w) + \frac{\lambda}{2}\|w\|^2$. Den Gradienten dazu können wir aber ohne Kenntnis von $\mathcal{D}$ nicht bestimmen. Wir gehen daher rundenweise vor, wählen in jeder Runde ein neues zufälliges Beispiel $(x, y) \sim \mathcal{D}$ und realisieren den nächsten Schritt des Gradientenabstiegs mit der in (4) spezifizierten Kostenfunktion. Beachte, dass $\mathbb{E}_{(x,y)\sim\mathcal{D}}[\|h_w(x) - y\|^2] = L_{\mathcal{D}}(h_w)$, d.h., im statistischen Mittel entspricht ℓ der idealen (aber uns unbekannten) Kostenfunktion. Verfahren dieser Bauart heißen „stochastische Gradientenabstiegsverfahren“.

Es erweist sich als vorteilhaft, die Schrittweite η des Gradientenabstiegs im Laufe der Iterationen zu reduzieren. Die in der i-ten Iteration verwendete Schrittweite bezeichnen wir mit η_i. Weiterhin ist es gut, den Gewichtsvektor nicht mit $\vec{0}$ zu initialisieren sondern mit einem zufällig gewählten Vektor (gemäß einer Verteilug, deren Realisierung i.A. Vektoren in der lokalen Umgebung von $\vec{0}$ produziert). Ein Startvektor $\vec{0}$ hätte nämlich zwei Nachteile:

- Falls zwei aufeinander folgende Schichten vollständig vernetzt sind, dann würden Gewichtsparameter, welche der selben Schicht angehören, auch nach Aktualisierung gleich bleiben (was die Freiheitsgrade zu sehr einschränkt).
- Mit identischen Startvektoren in verschiedenen Iterationen des Verfahrens wird, im Vergleich zu wechselnden Startvektoren, ein kleinerer Teil des Parameterraumes exploriert.

Es folgt der Pseudocode für das stochastische Gradientenabstiegsverfahren. Es verwendet die Hilfsprozedur „backpropagation“, die wir im Anschluss erläutern.

Parameter: Anzahl $\tau \geq 1$ an Iterationen, Schrittweiten $\eta_1, \ldots, \eta_\tau > 0$ und Regularisierungsparameter $\lambda > 0$

Eingabe: geschichtete FFNN-Architektur (V, E, σ)

Initialisierung: Wähle $w^{(1)}$ zufällig.

Hauptschleife: Für $i = 1, \ldots, \tau$ mache folgendes:

1. Nimm ein zufälliges Beispiel $(x, y) \sim \mathcal{D}$.
2. Berechne den Gradienten $G^{(i)} := \text{backpropagation}(V, E, \sigma, x, y, w^{(i)})$.
3. Setze $w^{(i+1)} = w^{(i)} - \eta_i(G^{(i)} + \lambda w^{(i)})$.

Ausgabe Gib einen Gewichtsvektor $\bar{w} \in \{w^{(1)}, \ldots, w^{(\tau)}\}$ aus, der auf einer Validierungsmenge am besten abschneidet.

Wir werden nachweisen, dass die Aktualisierung von w in Schritt 3 der Hauptschleife dem allgemeinen Schema des Gradientenabstiegs folgt, und zwar bezüglich der in (4) spezifizierten Kostenfunktion. Dabei ist der Term $\lambda w^{(i)}$ gerade der Gradient von $\frac{\lambda}{2}\|w\|^2$ an der Stelle $w^{(i)}$. Es wird also noch zu zeigen sein, dass der durch den Aufruf von Backpropagation berechnete Term $G^{(i)}$ den Gradienten von $\|h_w(x) - y\|^2 = \|o_T(x) - y\|^2$ an der Stelle $w^{(i)}$ bildet.[2]

Wir fassen im Folgenden $w : E \to \mathbb{R}$ als Vektor $w \in \mathbb{R}^E$ auf (mit einer Gewichtskomponente für jede Kante). Wir setzen der Einfachheit halber voraus, dass zwei aufeinander folgende Schichten vollständig vernetzt sind. Die allgemeine Prozedur ergibt sich aus der hier geschilderten, indem die Gewichtsparameter zu fehlenden Kanten stets auf den Wert 0 gesetzt werden.
Backpropagation ist die folgende Prozedur.

[2]Dieser Gradient lässt sich nicht analytisch, sondern nur algorithmisch, ermitteln.

Eingabe: geschichtete FFNN-Architektur (V, E, σ), gelabeltes Beispiel (x, y) und ein Gewichtsvektor $w \in \mathbb{R}^E$.

Initialisierung: 1. Kreiere die Schichten $V_0, V_1, \ldots, V_T$ mit $V_t = \{v_{t,1}, \ldots, v_{t,k_t}\}$.

2. Für $t = 0, \ldots, T-1$, $i = 1, \ldots, k_{t+1}$ und $j = 1, \ldots, k_t$ setze $W_{t,i,j} := w((v_{t,j}, v_{t+1,i}))$. W_t sei dann die resultierende $(k_{t+1} \times k_t)$-Matrix.

Vorwärtsphase: (Berechnung der Ausgabe des Netzwerks)

1. Es bezeichne $o_0 = (x, 1) \in \mathbb{R}^{n+1}$ den Vektor der Ausgabewerte der Eingabeschicht.
2. Für $t = 1, \ldots, T$ und $i = 1, \ldots, k_t$ setze

$$a_{t,i} := \sum_{j=1}^{k_{t-1}} W_{t-1,i,j} o_{t-1,j} \quad \text{und} \quad o_{t,i} := \sigma(a_{t,i}) \ .$$

$a_t, o_t \in \mathbb{R}^{k_t}$ seien dann die resultierenden Vektoren.

Rückwärtsphase: (Berechnung eines Gradienten)

1. Setze $\delta_T := o_T - y$.
2. Für $t = T-1, \ldots, 1$ und $i = 1, \ldots, k_t$ setze

$$\delta_{t,i} := \sum_{j=1}^{k_{t+1}} W_{t,j,i} \delta_{t+1,j} \sigma'(a_{t+1,j}) \ .$$

$\delta_t \in \mathbb{R}^{k_t}$ sein dann der resultierende Vektor.

Ausgabe: Gib $G \in \mathbb{R}^E$ aus mit

$$G(((v_{t-1,j}, v_{t,i}) := \delta_{t,i} \sigma'(a_{t,i}) o_{t-1,j} \ .$$

In der nun folgenden Analyse werden wir den Zusammenhang zum Gradientenabstieg herstellen.

Es sei $\ell_t(u)$ der (quadratische) Verlustwert, der bezüglich des Beispiels (x, y) entsteht, wenn u die Ausgabewerte der t-ten Schicht repräsentiert. Diese Werte ergeben sich rekursiv wie folgt:

1. $\ell_T(u) = \frac{1}{2}\|u - y\|^2$.
2. $\ell_t(u) = \ell_{t+1}(\sigma(W_t u))$ für $t = T-1, \ldots, 1$.

Aus dieser Rekursion ergeben sich die Funktionalmatrizen (bzw. Gradienten, da die Verlustfunktion Werte in $\mathbb{R} = \mathbb{R}^1$ annimmt) $J_u(\ell_T) = u - y$ und, für $t = T-1, \ldots, 1$ unter Anwendung der Kettenregel, $J_u(\ell_t) = J_{\sigma(W_t u)}(\ell_{t+1})\mathrm{diag}(\vec{\sigma'}(W_t u))W_t$. Setzen wir $u := o_t$ und $\delta_t := J_{o_t}(\ell_t)$, dann ergibt sich

$$\delta_T = o_T - y$$

und für $t = T-1, \ldots, 1$ gilt

$$\begin{aligned} \delta_t = J_{o_t}(\ell_t) &= J_{\sigma(W_t o_t)}(\ell_{t+1})\mathrm{diag}(\vec{\sigma'}(W_t o_t))W_t \\ &= J_{o_{t+1}}(\ell_{t+1})\mathrm{diag}(\vec{\sigma'}(a_{t+1})W_t = \delta_{t+1}\mathrm{diag}(\vec{\sigma'}(a_{t+1}))W_t \ . \end{aligned}$$

Diese rekursive Rechenvorschrift zur Berechnung der δ_t-Werte stimmt überein mit der Berechnung der $\delta_{t,i}$-Werte während der Rückwärtsphase von Backpropagation.

Da $w \in \mathbb{R}^E$ die programmierbaren Parameter sind, reicht es nicht aus die Gradienten $\delta_t = J_{o_t}(\ell_t)$ zu kennen. Vielmehr müssen wir die Funktion

$$g_t(W_{t-1}) := \ell_t(o_t) = \ell_t(\vec{\sigma}(a_t)) = \ell_t(\vec{\sigma}(W_{t-1}o_{t-1}))$$

betrachten, welche den auf Schicht t entstehenden Verlust als eine Funktion in W_{t-1} auffasst[3], um dann ihren Gradienten an der Stelle W_{t-1} zu bestimmen. Dies geschieht zunächst für $t = T$ und danach für $t = T-1, \ldots, 1$, so dass nach und nach der Gradient G für alle Gewichtsparameter bestimmt wird.

Zu diesem Zweck ist es besser, $W_{t-1} \in \mathbb{R}^{k_t \times k_{t-1}}$ als Vektor $w_{t-1} \in \mathbb{R}^{k_t k_{t-1}}$ aufzufassen. Sei also w_{t-1} der Vektor der entsteht, wenn die Zeilen von W_{t-1} aneinander gehängt werden. Weiter sei

$$O_{t-1} := \begin{pmatrix} o_{t-1}^\top & \vec{0} & . & . & . & \vec{0} \\ \vec{0} & o_{t-1}^\top & . & . & . & \vec{0} \\ . & . & . & & & . \\ . & . & & . & & . \\ . & . & & & . & . \\ \vec{0} & \vec{0} & . & . & . & o_{t-1}^\top \end{pmatrix} \in \mathbb{R}^{k_t \times k_t k_{t-1}} \ ,$$

so dass $W_{t-1}o_{t-1} = O_{t-1}w_{t-1}$. Wir erhalten dann die Gleichung

$$g_t(w_{t-1}) = \ell_t(\vec{\sigma}(O_{t-1}w_{t-1})) \ .$$

Anwendung der Kettenregel liefert

$$\begin{aligned} J_{w_{t-1}}(g_t) &= J_{\vec{\sigma}(O_{t-1}w_{t-1})}(\ell_t)\mathrm{diag}(\vec{\sigma'}(O_{t-1}w_{t-1}))O_{t-1} \\ &= J_{o_t}(\ell_t)\mathrm{diag}(\vec{\sigma'}(a_t))O_{t-1} \\ &= \delta_t\mathrm{diag}(\vec{\sigma'}(a_t))O_{t-1} \\ &= (\delta_{t,1}\sigma'(a_{t,1})o_{t-1}^\top, \ldots, \delta_{t,k_t}\sigma'(a_{t,k_t})o_{t-1}^\top) \ . \end{aligned}$$

[3]wobei alle anderen Gewichtsparameter als Konstanten betrachtet werden, wie das bei partiellen Ableitungen stets der Fall ist

Der Gradient $J_{w_{t-1}}(g_t)$ ist ein Vektor mit $k_t k_{t-1}$ Komponenten. Die Komponente, die für die Aktualisierung des Gewichtsparameters zur Kante $(v_{t-1,j}, v_{t,i})$ zuständig ist, finden wir in Position $(i-1)k_{t-1}+j$. Für den Gradientenvektor $G \in \mathbb{R}^E$ muss daher gelten: $G((v_{t-1,j}, v_{t,i}))$ stimmt überein mit der Komponente von $J_{w_{t-1}}(g_t)$ in Position $(i-1)k_{t-1}+j$. Inspektion von $(\delta_{t,1}\sigma'(a_{t,1})o_{t-1}^\top, \ldots, \delta_{t,k_t}\sigma'(a_{t,k_t})o_{t-1}^\top)$ liefert

$$G((v_{t-1,j}, v_{t,i})) = \delta_{t,i}\sigma'(a_{t,i})o_{t-1,j} \ .$$

Dies deckt sich aber mit der Art wie Backpropagation G in der Ausgabephase berechnet.

Résumée. Das in diesem Abschnitt angegebene Verfahren „Gradientenabstieg mit Verwendung von Backpropagation“ realisiert einen stochastischen Gradientenabstieg.