Sind künstliche neuronale Netzwerke herkömmlichen Methoden der Extrapolation überlegen?

Ole Petersen

Gymnasium Mainz-Oberstadt
Mathematik
Betreuende Lehrkraft: Frau Genz
Schuljahr 2018/2019
Mainz, den 6. Mai 2019

Inhaltsverzeichnis

Sind künstliche neuronale Netzwerke herkömmlichen Methoden der Extrapolation überlegen?

Ole Petersen

7. Dezember 2019

1 Problemstellung

In verschiedensten Fachgebieten wie Technik, Naturwissenschaften, Finanzwissenschaften oder Datenwissenschaften gibt es häufig die Aufgabe, aus gegebenen Daten Entwicklungen in der Zukunft zu prognostizieren. Standardmethoden hierfür sind Ausgleichsgeraden und -Polynome. Diese haben jedoch gewisse Beschränkungen. Im Folgenden soll untersucht werden, wie gut sich die in letzter Zeit im Bereich der künstlichen Intelligenz immer populärer gewordenen künstlichen neuronalen Netzwerke im Vergleich dazu für Extrapolationsanwendungen einsetzen lassen.

2 Was sind künstliche neuronale Netzwerke?

Künstliche neuronale Netzwerke sind ein Teil der künstlichen Intelligenz im Bereich des maschinellen Lernens. Sie bestehen aus vernetzten künstlichen Neuronen, die üblicherweise in Schichten angeordnet sind. Sie berechnen Ausgaben aus Eingaben, mathematisch können sie daher als vektorwertige Funktion mit Eingabevektor aufgefasst werden.

2.1 künstliche Neuronen

Künstliche Neuronen sind Objekte, die aus einem gegebenen Eingabevektor eine Ausgabe, die sogenannte „Aktivierung", berechnen, die ein Skalar ist.

Künstliche Neuronen summieren die Koordinaten ihres Eingabevektors gewichtet auf. Darauf kann noch eine Nullpunktverschiebung, das sogenannte Bias, addiert werden. Sie besitzen eine Aktivierungsfunktion, deren Funktionswert die Aktivierung des Neurons ist. Für die gewichtete Summe ist ein Gewichtungsvektor in der Dimension des Eingabevektors nötig. Bei einem Eingabevektor $\vec{e}$, einem Gewichtungsvektor $\vec{w}$, einer Aktivierungsfunktion f und einem Bias b würde für die Aktivierung a des Neurons also gelten:

$$\begin{aligned} a &= f(b + \sum_i w_i * e_i) \\ &= f(b + \vec{e} \cdot \vec{w}) \end{aligned}$$

2.2 Verknüpfung mehrerer Neuronen zu einem künstlichen neuronalen Netzwerk

Die Eingabevektoren der Neuronen können sich aus den Eingabewerten in das Netzwerk und den Aktivierungen anderer Neuronen zusammensetzen. Im Folgenden werde ich ein paar Begriffe zu möglichen Architekturen neuronalen Netzwerken klären:

2.2.1 Verbindung

Es existiert eine Verbindung von einem Neuron A zu einem Neuron B , wenn As Aktivierung eine Koordinate des Eingabevektors von B ist. Bei künstlichen Neuronen steht das Gewicht, mit dem die Aktivierung As bei B multipliziert wird, für das mathematische Äquivalent der „Stärke“ der Verbindung von A zu B. Verbindungen sind gerichtet, somit können künstliche neuronale Netzwerke als gerichtete Graphen aufgefasst werden, bei denen die Neuronen Knoten und die Verbindungen Kanten entsprechen.

2.2.2 feedforward-Netz

Wenn ein Kreis in einem neuronalen Netzwerk auftaucht, ist die Aktivierung der Neuronen im Kreis nicht mehr bestimmbar, weil sie gegenseitig voneinander abhängen. Um das zu vermeiden und den Aufbau anschaulich zu gestalten, werden die Neuronen üblicherweise in gerichteten Schichten angeordnet, bei denen nur Verbindungen von Neuronen einer Schicht zur denen der nächsten Schicht existieren. Diese Eigenschaft nennt man „feedforward”. Eingabe- und Ausgabeschicht erfüllen eine besondere Aufgabe: Die Aktivierung der Neuronen der Eingabeschicht entspricht dem Wert einer Koordinate des Eingabevektors und die Koordinaten des Ausgabevektors sind die Aktivierungen entsprechender Neuronen in der Ausgabeschicht.

Zusammengefasst ist der Ablauf der Berechnung einer Ausgabe bei einem feedforward-Netz also Folgender:

1. Die Aktivierung der Neuronen der Eingabeschicht wird je auf den Wert der entsprechenden Koordinate des Eingabevektors gesetzt.

2. Die Aktivierung der Neuronen der nächsten Schicht wird berechnet. Dies ist wegen der „feedforward“-Eigenschaft möglich. Die genaue Berechnung erfolgt wie in 2.1 erklärt. Dieser Schritt wird bei den nächsten Schichten wiederholt, bis die letzte Schicht, die Ausgabeschicht, erreicht ist.

3. Die Aktivierung der Neuronen der Ausgabeschicht entspricht je einer Koordinate des Ausgabevektors und kann nun eingesetzt werden. Dann wird der Ausgabevektor zurückgegeben.

2.2.3 hidden layer

Eine Schicht zwischen Ein- und Ausgabeschicht. Neuronale Netzwerke mit vielen hidden layers werden auch „deep neural networks“ genannt.

2.2.4 fully connected layer

Ein sogenanntes „fully connected layer“ kennzeichnet sich dadurch, dass eine Verbindung von jedem Neuron einer Schicht zu jedem Neuron der nächsten Schicht existiert. Solche Strukturen können schnell extrem rechenintensiv werden, wie kurz plausibilisiert werden soll: Angenommen, man wollte ein Bild mit einer Auflösung von zwölf Megapixeln analysieren. Die Eingabeschicht hätte dann $3*12*10^6$ Neuronen (es handelt sich um ein rgb-Farbbild). Wollte man direkt dahinter ein fully connected layer mit ebenso vielen Neuronen schalten, gäbe es von jedem Neuron der Eingabeschicht eine Verbindung zu jedem Neuron der nächsten Schicht, für deren Gewichtung jeweils für Gleitkommazahlen übliche 8 Byte Speicherplatz notwendig sind. Dann läge bereits der Speicherbedarf nur für die Gewichte des ersten hidden layers bei 8 Byte$*(3*12*10^6)^2 = 10.368*10^{15}$Byte$=$ 10.368 Petabyte. Vom Berechnen von Ausgaben oder gar vom Anlernen des Netzwerkes muss gar nicht erst gesprochen werden.

2.2.5 einschichtiges neuronales Netzwerk

Neuronales Netzwerk mit nur einer Schicht, der Ausgabeschicht. Die Eingabeschicht wird nicht mitgezählt, weil ihre Neuronen ihre Aktivierung nicht berechnen müssen, sondern diese direkt durch den Eingabevektor in das Netzwerk definiert wird.

2.2.6 mehrschichtige neuronales Netzwerk

Neuronale Netzwerke mit mehreren Schichten haben das Potential, komplexere Zusammenhänge als die einschichtigen Varianten 2.2.5 zu modellieren. Um diesen Vorteil nutzen zu können, dürfen die Aktivierungsfunktionen allerdings nicht linear sein. Beweis: Ich zeige, dass das Verhalten jedes mehrschichtigen neuronalen Netzwerks mit ausschließlich linearen Aktivierungsfunktionen dem eines einschichtigen neuronalen Netzwerks entspricht. Das geschieht durch vollständige Induktion über die Anzahl n der Schichten des multiple-layer artificial neural networks. Hierzu sei ein beliebiges n-schichtiges neuronales Netzwerk mit ausschließlich linearen Aktivierungsfunktionen gegeben.

Induktionsanfang: n=1 Ein KNN (künstliches neuronales Netzwerk) mit genau einer Schicht verhält sich wie ein einschichtiges neuronales Netzwerk, weil es eines ist.

Induktionsschritt Angenommen, jedes n-schichtige neuronale Netzwerk mit ausschließlich linearen Aktivierungsfunktionen verhält sich wie ein einschichtiges neuronales Netzwerk. Dann verhält sich auch jedes $n+1$-schichtige neuronale Netzwerk mit ausschließlich linearen Aktivierungsfunktionen wie ein einschichtiges neuronales Netzwerk, da es sich wie ein n-schichtiges neuronales Netzwerk mit ausschließlich linearen Aktivierungsfunktionen verhält. Um dies nachzuweisen betrachte ich die letzten drei Schichten des $n+1$-schichtigen Netzwerks. Sei das i-te Neuron der k-ten Schicht $n_{k,i}$, seine Aktivierung $a_{k,i}$ und die Gewichtung der Verbindung vom Neuron $n_{k-1,i}$ zum Neuron $n_{k,j}$ $w_{k,i,j}$. Außerdem sei die Aktivierungsfunktion des Neurons $n_{k,i}$ $f_{k,i}(s) := d_{k,i} * s + e_{k,i}$. Weiterhin sei die Anzahl der Neuronen in der k-ten Schicht des Netzwerkes r_k. Es gilt jetzt für alle natürlichen $m < r_{n+1}$:

$$a_{n+1,m} = f_{n+1,m}(\sum_{u=1}^{r_n} w_{n+1,u,m} * a_{n,u}) = f_{n+1,m}(\sum_{u=1}^{r_n} w_{n+1,u,m} * f_{n,u}(\sum_{g=1}^{r_{n-1}} w_{n,g,u} * a_{n-1,g})) =$$
$$d_{n+1,m} * (\sum_{u=1}^{r_n} w_{n+1,u,m} * (d_{n,u} * (\sum_{g=1}^{r_{n-1}} w_{n,g,u} * a_{n-1,g}) + e_{n,u})) + e_{n+1,m} =$$
$$d_{n+1,m} * (\sum_{u=1}^{r_n} (w_{n+1,u,m} * d_{n,u}) * (\sum_{g=1}^{r_{n-1}} w_{n,g,u} * a_{n-1,g})) + (e_{n+1,m} + \sum_{u=1}^{r_n} w_{n+1,u,m} * e_{n,u}) =$$
$$d_{n+1,m} * (\sum_{u=1}^{r_n} (\sum_{g=1}^{r_{n-1}} (w_{n+1,u,m} * d_{n,u}) * w_{n,g,u} * a_{n-1,g})) + (e_{n+1,m} + \sum_{u=1}^{r_n} w_{n+1,u,m} * e_{n,u}) =$$
$$d_{n+1,m} * (\sum_{g=1}^{r_{n-1}} (\sum_{u=1}^{r_n} (w_{n+1,u,m} * d_{n,u}) * w_{n,g,u} * a_{n-1,g})) + (e_{n+1,m} + \sum_{u=1}^{r_n} w_{n+1,u,m} * e_{n,u}) =$$
$$d_{n+1,m} * (\sum_{g=1}^{r_{n-1}} (\sum_{u=1}^{r_n} (w_{n+1,u,m} * d_{n,u} * w_{n,g,u})) * a_{n-1,g}) + (e_{n+1,m} + \sum_{u=1}^{r_n} w_{n+1,u,m} * e_{n,u})$$

Hier wurde gezeigt, dass die letzten beiden Schichten des Netzwerkes sich auch durch eine einzige darstellen lassen, nämlich dadurch, dass zunächst alle Verbindungen zwischen den letzten drei Schichten getrennt werden und neue Verbindungen von der $n-1$-ten Schicht zur $n+1$-ten Schicht geschaffen werden. Deren Gewichtung vom Neuron $n_{n-1,g}$ zum Neuron $n_{n+1,m}$ ist $\sum_{u=1}^{r_n} (w_{n+1,u,m} * d_{n,u} * w_{n,g,u})$. Außerdem muss das Bias der Aktivierungsfunktion des Neurons $n_{n+1,m}$ angepasst werden zu $e_{n+1,m} + \sum_{u=1}^{r_n} w_{n+1,u,m} * e_{n,u}$.

Mit diesen Änderungen verhält es sich wie ein n-schichtiges neuronales Netzwerk mit ausschließlich linearen Aktivierungsfunktionen, hat aber eine Schicht weniger. Das veränderte Netzwerk lässt sich laut Induktionsvoraussetzung als einschichtiges neuronales Netzwerk darstellen.

Hinweis: Im Fall n=1 sind mit allen Neuronen der nullten Schicht die Eingangsschichtneuronen gemeint.

3 Adaption für die Extrapolation

Bisher wurde nur geklärt, wie KNNs aus Eingaben Ausgaben generieren. Um sie für Extrapolationen nutzen zu können, werden nun ein paar Annahmen gemacht:

- Es soll eine Funktion $f : \mathbb{R} \to \mathbb{R}$ extrapoliert werden.
- Für die Eingabewerte x_i in das Netzwerk gilt für alle ganzzahligen $0 \leq i < n$, dass $x_i = f(c - i * \Delta x)$ mit $c, \Delta x \in \mathbb{R}$ ist, wobei Δx bildlich gesprochen der Abstand der Eingabepunkte in x-Richtung, n die Anzahl der Eingabewerte und c „die letzte bekannte Stelle“ von f ist. Das Netzwerk habe nur einen Ausgabewert y, der möglichst gut an $f(c + \Delta x)$ angenähert werden soll, das Ziel des Netzwerkes ist also „ den nächsten Funktionswert “ vorauszusagen. Dabei sind alle x_i nicht zu verwechseln mit den x-Werten der Punkte auf der Funktion im geometrischen Zusammenhang.

An dieser Stelle wird schon der erste Nachteil der hier betrachteten Form neuronaler Netzwerke klar: Es lassen sich nur Funktionswerte von Stellen, deren x-Wert sich darstellen lässt

als $c - i * \Delta x$, als Inputs verwenden und das Output ist nur der Funktionswert einer einzigen Stelle. Es lassen sich aber alle Funktionswerte an Stellen der Form $c + i * \Delta x$ extrapolieren, indem das selbe Netzwerk mehrmals hintereinander mit seiner letzten Ausgabe als zusätzliche Eingabe angewendet wird. Die genaue Funktionsweise dieses Verfahrens wird im Abschnitt 6 beleuchtet.

4 theoretische Fähigkeiten

In diesem Abschnitt soll geklärt werden, wozu optimal trainierte KNNs, d.h. KNNs mit optimal angepassten Parametern, fähig sind.

4.1 einschichtige neuronale Netze

Als erstes werden einschichtige neuronale Netzwerke mit $l + 1$ Eingabewerten und lediglich einem Ausgabeneuron mit der Aktivierung y mit der Identitätsfunktion als Aktivierungsfunktion betrachtet, bei denen das Gewicht vom i-ten Eingabeneuron mit dem Eingabewert $x_i = f(c - i * \Delta x)$ zum Ausgabeneuron w_i sei. Die Nullpunktverschiebung sei b. Der Ausgabewert ist also $y = b + \sum_{i=0}^{l} w_i * f(c - i * \Delta x)$.

4.1.1 ganzrationale Funktionen

Ich betrachte eine ganzrationale Funktion l-ten Grades. Im Folgenden soll gezeigt werden, dass das betrachtete KNN ganzrationale Funktionen beliebigen Grades sogar extrapolieren kann, ohne auf die spezielle Funktion trainiert zu werden, die Gewichte müssen nur dem Grad des Polynoms angepasst sein. Nun soll gezeigt werden, dass und wie diese Menge von Funktionen mit dem gegebenen Netzwerk exakt extrapolierbar ist.

Idee für die Werte der Gewichte Um die Behauptung zu zeigen, muss als erstes eine Vermutung aufgestellt werden, mit welchen Gewichtswerten das neuronale Netzwerk korrekt arbeitet. Hierzu gehe ich induktiv über den Grad des Polynoms vor. Ein Polynom des Grades null, eine Konstante, lässt sich problemlos dadurch extrapolieren, den einzig nötigen Eingabewert x_0 direkt als Ausgabewert zurückzugeben. Wenn der Grad des Polynoms von m zu $m+1$ erhöht werden soll, muss ein Term dafür ergänzt werden, dass die $m + 1$-te Ableitung der Funktion nicht null sein muss. Das liefert folgende Lösungsidee:

- Grad 0: $y(x_0) = x_0$
- Grad 1: $y(x_0, x_1) = x_0 + (x_0 - x_1) = 2x_0 - x_1$, Änderung des y-Wertes muss zusätzlich berücksichtigt werden
- Grad 2: $y(x_0, x_1, x_2) = x_0 + (x_0 - x_1) + ((x_0 - x_1) - (x_1 - x_2)) = 3x_0 - 3x_1 + x_2$, Änderung der Änderung des y-Wertes muss zusätzlich berücksichtigt werden
- Grad 3: $y(x_0, x_1, x_2, x_3) = x_0 + (x_0 - x_1) + ((x_0 - x_1) - (x_1 - x_2)) + (((x_0 - x_1) - (x_1 - x_2)) - ((x_1 - x_2) - (x_2 - x_3))) = 4x_0 - 6x_1 + 4x_2 - x_3$, Änderung der Änderung der Änderung des y-Wertes muss zusätzlich berücksichtigt werden

Eine genaue Betrachtung der Koeffizienten liefert für ein Polynom des Grades l dann folgende Vermutung:

$$w_i = \binom{l+1}{i+1} * (-1)^i$$

Nun soll bewiesen werden, dass ein neuronales Netzwerk mit diesen Gewichten ein Polynom eines Grades von maximal l exakt extrapoliert.

Beweis Sei $f(r) := \sum_{m=0}^{l} a_m * r^m$ mit reellen a, r die zu extrapolierende Funktion, also eine ganzrationale Funktion l-ten Grades. Das Netzwerk sei das in Abschnitt 4.1 beschriebene Netzwerk mit $l+1$ Eingabewerten, $w_i := \binom{l+1}{i+1} * (-1)^i$ und $b := 0$. Zu zeigen ist dann, dass $y = f(c + \Delta x)$. Dann gilt:

$$y = \sum_{i=0}^{l} w_i * x_i = \sum_{i=0}^{l} \binom{l+1}{i+1} * (-1)^i * x_i = \sum_{i=0}^{l} \binom{l+1}{i+1} * (-1)^i * f(c - i * \Delta x) =$$

$$\sum_{i=0}^{l} (\binom{l+1}{i+1} * (-1)^i * \sum_{m=0}^{l} (a_m * (c - i * \Delta x)^m)) =$$

Anwendung des binomischen Lehrsatzes

$$\sum_{i=0}^{l} (\binom{l+1}{i+1} * (-1)^i * \sum_{m=0}^{l} (a_m * \sum_{k=0}^{m} (\binom{m}{k} * c^k * (-i * \Delta x)^{m-k})))$$

Nun wird die Summe als Polynom in c geschrieben. Dazu wird die Summe umsortiert, sodass alle Summanden mit einer gleichen Potenz p von c hintereinander stehen und dann wird aus diesem „Block“ c^p ausgeklammert. Für ein bestimmtes p ist in der Summe über k nur der Summand $k = p$ relevant. Deshalb sind bei der Summe über m nur die Summenden relevant, bei denen $p \leq m$ gilt, weil bei allen anderen Werten von m der Faktor c^p in der Summe über k gar nicht auftaucht. Bei der Summe über i sind alle Summanden relevant.

Nach diesem Verfahren werden für alle p die Koeffizienten von c^p bestimmt. Die umsortierte Summe sieht dann folgendermaßen aus:

$$y = \sum_{p=0}^{l} (\sum_{i=0}^{l} (\binom{l+1}{i+1} * (-1)^i * (\sum_{m=p}^{l} (a_m * \binom{m}{p} * (-i * \Delta x)^{m-p}))) * c^p)$$

Jetzt soll der Beweis von der anderen Seite aus weitergeführt werden, nämlich von dem gewünschten Output des Netzwerkes aus. Das ist:

$$f(c + \Delta x) = \sum_{i=0}^{l} a_i * (c + \Delta x)^i) = \sum_{i=0}^{l} a_i * (\sum_{j=0}^{i} \binom{i}{j} * c^j * \Delta x^{i-j})$$

Auch diese Summe wird jetzt als Polynom in c geschrieben. Ein Faktor von genau c^p taucht hierbei nur auf beim Summanden $j = p$ auf. Damit dieser vorkommt, muss $p \leq i$ gelten. Damit gilt:

$$f(c + \Delta x) = \sum_{p=0}^{l} ((\sum_{i=p}^{l} a_i * \binom{i}{p} * \Delta x^{i-p}) * c^p)$$

Weil zwei Polynome gleich sind, wenn ihre Koeffizienten gleich sind, gilt die zu zeigende Aussage $y = f(c + \Delta x)$, wenn

$$\sum_{i=0}^{l} (\binom{l+1}{i+1} * (-1)^i * (\sum_{m=p}^{l} (a_m * \binom{m}{p} * (-i * \Delta x)^{m-p}) = \sum_{i=p}^{l} a_i * \binom{i}{p} * \Delta x^{i-p}$$

Nun werden in einem Schritt sowohl die linke als auch die rechte Seite der Gleichung als Polynom in Δx geschrieben. Auf beiden Seiten stehen nur Potenzen von Δx von null bis $l - p$, weshalb die äußere Summe nur diesen Bereich abdeckt.

Auf der linken Seite in der Summe über alle m tauchen Faktoren von genau Δx^q nur beim Faktor $m = q + p$ auf und das bei allen i.

Auf der rechten Seite wird lediglich i durch $q := i+p$ substituiert. Dadurch „beginnt“ und „endet“ die Summe p früher.

$$\Leftrightarrow \sum_{q=0}^{l-p}((\sum_{i=0}^{l}(\tbinom{l+1}{i+1} * (-1)^i * (-i)^q * a_{q+p} * \tbinom{q+p}{p}))) * \Delta x^q) = \sum_{q=0}^{l-p}((a_{q+p} * \tbinom{q+p}{p})) * \Delta x^q)$$

Weil diese Gleichungen wieder Polynome in der selben Variable, dieses mal Δx sind, sind sie gleich, wenn ihre Koeffizienten gleich sind. Die Aussage gilt somit, wenn die folgende Zeile gilt:

$$\sum_{i=0}^{l}(\tbinom{l+1}{i+1} * (-1)^i * (-i)^q * a_{q+p} * \tbinom{q+p}{p})) = a_{q+p} * \tbinom{q+p}{p} \qquad /(a_{q+p}\tbinom{q+p}{p}))$$

$$\Leftarrow \sum_{i=0}^{l}(\tbinom{l+1}{i+1} * (-1)^i * (-i)^q) = 1 \qquad \text{Substitution von } i \text{ durch } i-1$$

$$\Leftrightarrow \sum_{i=1}^{l+1}(\tbinom{l+1}{i} * (-1)^{i-1} * (-(i-1))^q) = 1$$

$$\Leftrightarrow \sum_{i=0}^{l+1}(\tbinom{l+1}{i} * (-1)^{i-1} * (-(i-1))^q) - \tbinom{l+1}{0} * (-1)^{0-1} * (-(0-1)) = 1 \qquad \text{Addition und gleichzeitige Subtraktion des nullten Summanden der Summe}$$

$$\Leftrightarrow \sum_{i=0}^{l+1}(\tbinom{l+1}{i} * (-1)^{i-1} * (-(i-1))^q) = 0$$

$$\Leftrightarrow \sum_{i=0}^{l+1}\tbinom{l+1}{i} * (-1)^i * (i-1)^q = 0$$

Diese Aussage wird nun durch die folgenden drei Lemmas bewiesen.

Lemma 4.1 *Es gilt* $\sum_{i=0}^{l}\tbinom{l}{i} * (-1)^i * \frac{i!}{(i-q)!} = 0$ *für alle natürlichen* l *und* q *mit* $q < l$.

Dies gilt, da

$$(1+x)^l = \sum_{i=0}^{l}\tbinom{l}{i} * x^i \qquad \frac{\mathrm{d}^q}{\mathrm{d}^q x}$$

$$\Rightarrow \frac{l!}{(l-q)!}(1+x)^{l-q} = \sum_{i=0}^{l}\tbinom{l}{i}\frac{i!}{(i-q)!}x^{i-q} \qquad -1 \text{ für x einsetzen}$$

$$\Rightarrow 0 = \sum_{i=0}^{l}\tbinom{l}{i}\frac{i!}{(i-q)!}(-1)^{i-q} \qquad /((-1)^{-q})$$

$$\Leftrightarrow 0 = \sum_{i=0}^{l}\tbinom{l}{i} * (-1)^i * \frac{i!}{(i-q)!}$$

Lemma 4.2 *Es gilt* $\sum_{i=0}^{l}\tbinom{l}{i}(-1)^i * i^p = 0$ *für alle natürlichen* l *und* p *mit* $p < l$.

Dies zeige ich induktiv über p:

Induktionsanfang: p=0 Einsetzen von $q = 0$ in das vorherige Lemma liefert:

$$\sum_{i=0}^{l}\binom{l}{i} * (-1)^i * \frac{i!}{(i-0)!} = \sum_{i=0}^{l}\binom{l}{i} * (-1)^i * i^0 = 0$$

Induktionsschritt von p zu $p+1$ Ich setze $q = p+1$ in das vorherige Lemma ein. Nun seien alle e_o so definiert, dass $\frac{i!}{(i-(p+1))!} = \sum_{o=0}^{p+1} e_o * i^o$. Dies ist möglich, da die höchste Potenz von i in $\frac{i!}{(i-(p+1))!}$ $p+1$ ist und der gesamte Term als Polynom darstellbar ist. Es gilt also laut Lemma 4.1 mit $q = p+1$:

$$\begin{aligned}
&\sum_{i=0}^{l} \binom{l}{i} * (-1)^i * \frac{i!}{(i-(p+1))!} = 0 \\
\Leftrightarrow &\sum_{i=0}^{l} \binom{l}{i} * (-1)^i * \sum_{o=0}^{p+1} e_o * i^o = 0 \\
\Leftrightarrow &\sum_{o=0}^{p+1} e_o * \sum_{i=0}^{l} \binom{l}{i} * (-1)^i * i^o = 0 \\
\Leftrightarrow &\sum_{o=0}^{p} e_o * 0 + \sum_{i=0}^{l} \binom{l}{i} * (-1)^i * i^{p+1} = 0 \\
\Leftrightarrow &\sum_{i=0}^{l} \binom{l}{i} * (-1)^i * i^{p+1} = 0
\end{aligned}$$

Lemma 4.3 *Es gilt* $\sum_{i=0}^{l} \binom{l}{i}(-1)^j * P(i) = 0$ *für alle natürlichen* l *und ganzrationale Funktionen* $P(x) := \sum_{p=0}^{l-1} a_p * x^p$ *mit einem Grad kleiner als* l*.*

Es gilt laut Lemma 4.2 :

$$\sum_{p=0}^{l-1} a_p * \sum_{i=0}^{l} \binom{l}{i}(-1)^i * i^p = \sum_{p=0}^{l-1} a_p * 0 = 0 = \sum_{i=0}^{l} \binom{l}{i}(-1)^i * \sum_{p=0}^{l-1} a_p * i^p = \sum_{i=0}^{l} \binom{l}{i}(-1)^i * P(i)$$

Setzt man in das letzte Lemma $l := l+1$ und $P(x) := (x-1)^q$ ein, so ergibt sich die zu zeigende Aussage.

Dieses Ergebnis zeigt, dass schon einschichtige KNNs in der Lage sind, ganzrationale Funktionen beliebigen Grades exakt zu extrapolieren. Dafür müssen sie nicht einmal auf eine spezifische Funktion trainiert werden: Es genügt, wenn sie gelernt haben, ein Polynom von maximal einem bestimmten Grad zu extrapolieren.

4.1.2 Exponentialfunktionen

Exponentialfunktionen lassen sich bekanntlich per Taylorreihe als Polynom darstellen. Das Problem daran ist, dass für eine exakte Prognose der Grad des Polynoms gegen unendlich gehen muss, was in der Praxis natürlich nicht möglich ist. Ist das Netzwerk hingegen auf eine spezifische Funktion optimiert, kann es sie trotzdem exakt extrapolieren.

Beweis Gegeben sei die zu extrapolierende Funktion $f(r) := a + e^{q*r}$ mit reellen a und q und ein neuronales Netzwerk wie in Abschnitt 3 beschrieben mit nur einem Eingabeneuron. Das Problem ist gelöst, wenn sich immer Werte für w_0 und b finden lassen,

sodass $f(c + \Delta x) = y$. Wenn $w_0 = e^{q*\Delta x}$ und $b = a * (1 - e^{q*\Delta x})$, gilt offenbar:

$$\begin{aligned} & f(c + \Delta x) = y \\ \Leftrightarrow\ & a + e^{q*(c+\Delta x)} = b + w_0 * (a + e^{q*c}) \\ \Leftrightarrow\ & a + e^{q*(c+\Delta x)} = a * (1 - e^{q*\Delta x}) + e^{q*\Delta x} * (a + e^{q*c}) \\ \Leftrightarrow\ & a + e^{q*(c+\Delta x)} = a - a * e^{q*\Delta x} + a * e^{q*\Delta x} + e^{q*\Delta x} * e^{q*c} \\ \Leftrightarrow\ & a + e^{q*(c+\Delta x)} = a + e^{q*(c+\Delta x)} \end{aligned}$$

4.1.3 Sinusfunktionen

Da Sinusfunktionen sich mithilfe von Taylorreihen beliebig genau darstellen lassen und die betrachteten Netzwerke beliebig große Polynome extrapolieren können, können sie auch beliebig genau Sinusfunktionen extrapolieren.

4.2 mehrschichtige Netzwerke

Laut dem sogenannten „universal approximation theorem“ können neuronale Netzwerke mit bestimmten nichtlinearen Aktivierungsfunktionen und einem hidden layer mit beliebig vielen Neuronen jede stetige Funktion beliebig genau approximieren. Deshalb werden in der Praxis auch tiefere neuronale Netzwerke eingesetzt. Die Faustregel ist: je tiefer das Netzwerk, desto komplexere Zusammenhänge kann es lernen.

5 Wie lernen sie?

Bisher wurde zwar gezeigt, dass künstliche neuronale Netzwerke bei optimaler Anpassung der Gewichte und perfektem Aufbau sehr mächtig sind, eine manuelle Eingabe der Gewichte ist jedoch in der Praxis nicht das Ziel. Das Netzwerk soll selbstständig lernen, aus Eingaben passende Ausgaben zu generieren. Dazu sind Lerndaten notwendig. Ein Lerndatensatz besteht aus Eingabewerten und den gewünschten zugehörigen Ausgabewerten.

5.1 Ein Optimierungsproblem

Um eine Optimierung möglich zu machen, muss zunächst klargestellt werden, was überhaupt eine gute Ausgabe ist. Hierzu wird eine sogenannte Fehlerfunktion E aufgestellt. Sie soll bewerten, wie gut das Netzwerk bei einem gegebenen Datensatz gearbeitet hat. Dazu werden alle Eingaben x in das Netzwerk, die gewünschten Ausgaben t und die tatsächlichen Ausgaben y des Netzwerks benötigt. Eine etablierte Methode für das Bilden der Fehlerfunktion ist, die quadrierten Abweichungen der Outputs vom Sollwert aufzusummieren und zu halbieren, also:

$$E = \frac{1}{2} \sum_i (t_i - y_i)^2$$

Je kleiner der Wert von E ist, desto besser hat das Netzwerk funktioniert. Beispielsweise sind gewünschte und tatsächliche Ausgabewerte offenbar identisch, wenn $E = 0$ ist. Der Vorteil einer solchen Fehlerfunktion ist, dass sich mit ihr das zuvor riesig scheinende Problem des

maschinellen Lernens auf ein Optimierungsproblem reduziert: Die Summe der Fehlerwerte für alle Lernbeispiele muss minimiert werden, indem die Gewichte angepasst werden. Bei k vorhandenen Lernbeispielen mit einer Fehlerfunktion E_i beim i-ten Beispiel und m im Netzwerk vorhandenen anpassbaren Parametern (Gewichten und Nullpunktverschiebungen), die in einen Vektor $\vec{q}$ geschrieben seien, muss also die Funktion

$$E_{ges} : \mathbb{R}^m \mapsto \mathbb{R}$$
$$E_{ges}(\vec{q}) := \sum_{i=1}^{k} E_i(\vec{q})$$

minimiert werden.

5.2 Gradientenverfahren

Ein Standardverfahren für solche Minimierungsprobleme ist das Gradientenverfahren. Es wird mit einem beliebigen Wert von $\vec{q}$ begonnen und der Fehlerwert iterativ verringert. Der Iterationsschritt ist:

$$\vec{q}_{n+1} = \vec{q}_n - \alpha * \begin{pmatrix} \frac{\partial E}{\partial q_1} \\ \frac{\partial E}{\partial q_2} \\ ... \\ \frac{\partial E}{\partial q_m} \\ \end{pmatrix}$$

Bildlich gesprochen ist E eine$m$-dimensionale „Fläche“ in einem $m+1$-dimensionalen Raum und das Problem, ein möglichst tief gelegener Ort zu finden, wobei $E(\vec{q})$ der Höhe an einer bestimmten Stelle $\vec{q}$ entspricht. Das Lernverfahren nutzt aus, dass der „steilste Weg nach unten“ an einer bestimmten Stelle in die Richtung des Gegenvektors des Gradienten von E an einer bestimmten Stelle ist. Die sogenannte „Lernkonstante“ α gibt dabei an, wie weit in die Richtung des schnellsten Abstiegs gegangen werden soll, also wie schnell optimiert wird. α kann aber nicht beliebig groß gewählt werden, weil es sonst zu einer Art sich aufschaukelnden Schwingung kommen kann, wenn der neue Wert von $\vec{q}$ „auf der anderen Seite des Tals“ liegt, siehe 5.4.

5.3 backpropagation

Dieser Gradient lässt sich numerisch bestimmen. Das ist aber rechenaufwendig und ungenau: Es muss für jedes Gewicht der Wert desselben minimal verändert und der Fehlerwert neu berechnet werden. Das würde das Verfahren praktisch unbrauchbar machen. Für feedforward-Netzwerke mit einfach differenzierbaren Aktivierungsfunktionen gibt es glücklicherweise ein effizienteres Verfahren namens Backpropagation, um den Gradienten zu berechnen. Um es zu erklären, müssen zunächst die Gegebenheiten genau definiert werden:

- Die Aktivierung des i-ten Neurons der k-ten Schicht sei $o_{k,i}$ und sein Bias $b_{k,i}$
- Der Eingabewert in die einfach differenzierbare Aktivierungsfunktion $f_{k,i} : \mathbb{R} \to \mathbb{R}$ des

i-ten Neurons der k-ten Schicht sei $net_{k,i}$

- Die nullte Schicht sei die Eingabeschicht und die $l-1$-te Schicht die Ausgabeschicht des Netzwerks. x_i sei der i-te Eingabewert und y_i der i-te Ausgabewert des Netzwerks. Damit gilt: $net_{0,i} = x_i$ und $y_i = o_{l-1,i}$
- Das Gewicht der Verbindung vom i-ten Neuron der k-ten Schicht zum j-ten Neurons der $k+1$-ten Schicht sei $w_{k,i,j}$
- Das gewünschte i-te Output des aktuellen Trainingsbeispiels sei t_i
- Die Fehlerfunktion sei definiert als $E := \frac{1}{2}\sum_n (t_n - o_{l-1,n})^2$

Nun wird mit einem Trick vorgegangen: zunächst wird für jedes Neuron iterativ über die Schichten von der Ausgabeschicht aus der Wert von $\frac{\mathrm{d}E}{\mathrm{d}o_{k,i}}$ berechnet. Daraus lässt sich dann $\frac{\mathrm{d}E}{\mathrm{d}w_{k,i,j}}$ berechnen. Das genaue Vorgehen wird im Folgenden hergeleitet:

5.3.1 Bestimmung von $\frac{\mathrm{d}E}{\mathrm{d}o_{k,i}}$

Iterationsstart Dieser wird durch die Ausgabeschicht realisiert. Dort gilt laut Kettenregel:

$$\frac{\mathrm{d}E}{\mathrm{d}o_{l-1,i}} = \frac{\mathrm{d}\frac{1}{2}\sum_n (t_n - o_{l-1,n})^2}{\mathrm{d}o_{l-1,i}} = \frac{\mathrm{d}\frac{1}{2}(t_i - o_{l-1,i})^2}{\mathrm{d}o_{l-1,i}} = \frac{\mathrm{d}\frac{1}{2}(t_i - o_{l-1,i})^2}{\mathrm{d}t_i - o_{l-1,i}} * \frac{\mathrm{d}t_i - o_{l-1,i}}{\mathrm{d}o_{l-1,i}} = o_{l-1,i} - t_i$$

Iterationsschritt Auch hier wird die Kettenregel angewendet. Jetzt wird allerdings E als Funktion mehrerer Variablen gesehen, nämlich als Funktion der Aktivierungen der Neuronen der Schicht $k+1$. Deshalb muss die Kettenregel für Funktionen mehrerer Variablen verwendet werden:

$$\begin{aligned}\frac{\mathrm{d}E}{\mathrm{d}o_{k,i}} = \sum_n \frac{\mathrm{d}E}{\mathrm{d}o_{k+1,n}} * \frac{\mathrm{d}o_{k+1,n}}{\mathrm{d}o_{k,i}} &= \sum_n \frac{\mathrm{d}E}{\mathrm{d}o_{k+1,n}} * \frac{\mathrm{d}o_{k+1,n}}{\mathrm{d}net_{k+1,n}} * \frac{\mathrm{d}net_{k+1,n}}{\mathrm{d}o_{k,i}} \\ &= \sum_n \frac{\mathrm{d}E}{\mathrm{d}o_{k+1,n}} * f'_{k+1,n}(net_{k+1,n}) * w_{k,i,n}\end{aligned}$$

5.3.2 Bestimmung von $\frac{\mathrm{d}E}{\mathrm{d}w_{k,i,j}}$ aus $\frac{\mathrm{d}E}{\mathrm{d}o_{k,i}}$

Es gilt laut der Kettenregel:

$$\frac{\mathrm{d}E}{\mathrm{d}w_{k,i,j}} = \frac{\mathrm{d}E}{\mathrm{d}o_{k+1,j}} * \frac{\mathrm{d}o_{k+1,j}}{\mathrm{d}net_{k+1,j}} * \frac{\mathrm{d}net_{k+1,j}}{\mathrm{d}w_{k,i,j}} = \frac{\mathrm{d}E}{\mathrm{d}o_{k+1,j}} * f'_{k+1,j}(net_{k+1,j}) * \frac{\mathrm{d}b_{k+1,j} + \sum_n o_{k,n} * w_{k,n,j}}{\mathrm{d}w_{k,i,j}} =$$
$$\frac{\mathrm{d}E}{\mathrm{d}o_{k+1,j}} * f'_{k+1,j}(net_{k+1,j}) * \frac{\mathrm{d}o_{k,i} * w_{k,i,j}}{\mathrm{d}w_{k,i,j}} = \frac{\mathrm{d}E}{\mathrm{d}o_{k+1,j}} * f'_{k+1,j}(net_{k+1,j}) * o_{k,i}$$

5.3.3 Bestimmung von $\frac{\mathrm{d}E}{\mathrm{d}b_{k,i}}$ aus $\frac{\mathrm{d}E}{\mathrm{d}o_{k,i}}$

Es gilt laut der Kettenregel:

$$\frac{\mathrm{d}E}{\mathrm{d}b_{k,i}} = \frac{\mathrm{d}E}{\mathrm{d}o_{k,i}} * \frac{\mathrm{d}o_{k,i}}{\mathrm{d}b_{k,i}} = \frac{\mathrm{d}E}{\mathrm{d}o_{k,i}} * f\prime_{k,i}(net_{k,i})$$

5.4 Erweiterungen und Optimierungen

mehrere Trainingsbeispiele Mit den bisher vorgestellten Methoden lässt sich nur der Gradient zu einem Trainingsbeispiel bestimmen, es soll aber der Fehlerwert zu allen Trainingsbeispielen minimiert werden. Es gilt für jeden Parameter, also für jedes Gewicht und Bias, p des Netzwerkes :

$$\frac{\mathrm{d}E_{ges}}{\mathrm{d}p} = \frac{\mathrm{d}\sum\limits_{n} E_n}{\mathrm{d}p} = \sum_{n} \frac{\mathrm{d}E_n}{\mathrm{d}p}$$

Es können also problemlos die einzelnen Gradienten aufsummiert bzw. ein Durchschnittswert des Gradienten über alle Trainingsbeispiele gebildet und dann das Gradientenverfahren angewendet werden.

Einteilung der Trainingsbeispiele Es gibt unterschiedliche Ansätze, aus wie vielen Trainingsbeispielen der Fehlergradient bestimmt wird, um die Parameter des Netzwerks zu aktualisieren. Die Bandbreite reicht von einem über einen Bruchteil der Trainingsbeispiele hin zu allen Trainingsbeispielen. Welche Methode am besten funktioniert, muss ausprobiert werden.

Bestimmung der Lernkonstante Abhängig von der Größenordnung der Eingabewerte in das Netzwerk muss die Lernkonstante angepasst werden. Sie bestimmt maßgeblich die Lerngeschwindigkeit des Netzwerks und ob die Gewichtswerte überhaupt konvergieren. Sie für jedes Beispiel manuell anzupassen ist umständlich. Ein Ansatz ist, mithilfe der Krümmung der Fehlerfunktion und ihrer Steigung am Startpunkt in die Richtung der stärksten Steigung den Abstand zu einem Minimum in der Richtung abzuschätzen und die Lernkonstante so auswählen, dass in einem Iterationsschritt ein bestimmter Anteil dieser Strecke zurückgelegt wird. Es haben sich bei mir Größenordnung von 0.01 bewährt. Eine Veränderung der Lernkonstante während des Lernens hat bei mir hingegen in der Regel dazu geführt, dass sich die Gewichte „aufschaukeln“ und schließlich ins Unendliche gehen.

Konzeption des Netzwerkes Wenn ein neuronales Netzwerk für eine bestimmte Anwendung entworfen werden soll, ist es sinnvoll, sich zu überlegen, wie komplex das Netzwerk mindestens sein muss, um die Aufgabe zu bewältigen. Wenn ein solches Netzwerk gefunden ist, sollte es noch etwas komplexer werden, weil das Lernverfahren normalerweise keine optimale Lösung findet. Es sollte aber auch nicht zu viele Parameter besitzen, weil sonst ein Effekt namens „overfitting“ auftritt: Das Netzwerk „lernt die Trainingsbeispiele auswendig“, statt das System hinter ihnen zu erkennen, wodurch es mit neuen Aufgaben überfordert ist.

6 eigenes Beispiel

Nun soll ein eigenes neuronales Netzwerk zur Extrapolation erstellt werden. Als Eingabe erhält es die letzten n Funktionswerte einer zu extrapolierenden Funktion $f := \mathbb{R} \to \mathbb{R}$ und

es soll darauf trainiert werden, die letzten $n-1$ Funktionswerte und eine möglichst gute Prognose für den nächsten Funktionswert auszugeben. Es mag zunächst unsinnig scheinen, die letzten $n-1$ Funktionswerte auch auszugeben, schließlich müssen sie lediglich kopiert werden. So lässt sich aber das Netzwerk unkompliziert beliebig oft hintereinander setzen, wodurch sich Prognosen für Funktionswerte, die weiter voraus sind, bestimmen lassen. Um den Rechenaufwand zu reduzieren, soll für eine Instanz des Netzwerks nur ein einschichtiges neuronales Netzwerk mit der Identitätsfunktion als Aktivierungsfunktion verwendet werden. Es muss logischerweise n Ein-und Ausgabeneuronen besitzen. Wenn das Netzwerk als Eingabewerte $a_0, a_1, ..., a_{n-1}$ erhält, sei seine Ausgabe:

$$(b + \sum_{i=0}^{n-1} w_i * a_i), a_0, a_1, ..., a_{n-2}$$

Die Eingaben in die erste Instanz des Netzwerks seien analog zu Abschnitt 3 die letzten n bekannten Funktionswerte $x_i = f(c - i * \Delta x)$, wobei x_0 der letzte bekannte Funktionswert und die nullte Eingabe in das Netzwerk sei. Das Netzwerk werde k Mal hintereinander gesetzt. Außerdem sei der nullte Ausgabewert der i-ten Instanz des Netzwerks y_i, wobei die Zählung der Instanzen bei eins beginne. Das Ziel ist, dass alle y_i möglichst gut mit dem Zielwert t_i übereinstimmen , der der tatsächliche i-te Funktionswert der zu extrapolierenden Funktion hinter dem letzten in das Netzwerk eingegebenen Wert ist, also $t_i = f(c + i * \Delta x)$. Der Fehlerwert für den i-ten extrapolierten Wert des Netzwerks E_i sei definiert als $E_i := \frac{1}{2}(t_i - y_i)^2$. Weil aber der Fehlerwert aller k Ausgaben minimiert werden soll, muss noch ein Gesamtfehlerwert definiert werden als $E_{ges} := \sum_{i=1}^{k} g_i * E_i$ mit reellen, positiven g_i als Gewichtung. Diese ist sinnvoll, da es in bestimmten Anwendungen wichtig sein könnte, dass gewisse Bereiche besonders treffend extrapoliert werden.

6.1 Bestimmung der Ausgaben des Netzwerks

Aufgrund der Struktur des Netzwerks gilt:

$$y_i = b + \sum_{j=0}^{n-1} w_j * a_j$$

mit $a_j = \left\{ \begin{array}{ll} y_{i-j-1} & \text{, wenn } j+1 < i \\ x_{j-i+1} & \text{, wenn } j+2 > i \end{array} \right\}$(siehe Abbildung 2). So lassen sich iterativ über i die Ausgaben des Netzwerks bestimmen.

6.2 Bestimmung der Gradienten

Zur effizienteren Berechnung gehe ich in mehreren Schritten vor:

6.2.1 Bestimmung von $\frac{\mathrm{d}y_i}{\mathrm{d}w_m}$

$$\frac{\mathrm{d}y_i}{\mathrm{d}w_m} = \frac{\mathrm{d}b + \sum_{j=0}^{n-1} w_j * a_j}{\mathrm{d}w_m} = \frac{\mathrm{d}b}{\mathrm{d}w_m} + \sum_{j=0}^{n-1} \frac{\mathrm{d}w_j * a_j}{\mathrm{d}w_m} = \sum_{j=0}^{n-1} \frac{\mathrm{d}w_j * a_j}{\mathrm{d}w_m}$$

Dabei gilt:

$$\frac{\mathrm{d}w_j * a_j}{\mathrm{d}w_m} = \left\{ \begin{array}{ll} \frac{\mathrm{d}w_j * y_{i-j-1}}{\mathrm{d}w_m} & \text{, wenn } j+1 < i \\ \frac{\mathrm{d}w_j * x_{j-i+1}}{\mathrm{d}w_m} & \text{, wenn } j+2 > i \end{array} \right\} = \left\{ \begin{array}{ll} w_j * \frac{\mathrm{d}y_{i-j-1}}{\mathrm{d}w_m} & \text{, wenn } j+1 < i \text{ und } j \neq m \\ \frac{\mathrm{d}w_m * y_{i-j-1}}{\mathrm{d}w_m} & \text{, wenn } j+1 < i \text{ und } j = m \\ \frac{\mathrm{d}w_m * x_{j-i+1}}{\mathrm{d}w_m} & \text{, wenn } j+2 > i \text{ und } j = m \\ 0 & \text{, wenn } j+2 > i \text{ und } j \neq m \end{array} \right\} =$$

$$\left\{ \begin{array}{ll} w_j * \frac{\mathrm{d}y_{i-j-1}}{\mathrm{d}w_m} & \text{, wenn } j+1 < i \text{ und } j \neq m \\ w_m * \frac{\mathrm{d}y_{i-j-1}}{\mathrm{d}w_m} + \frac{\mathrm{d}w_m}{\mathrm{d}w_m} * y_{i-j-1} = w_m * \frac{\mathrm{d}y_{i-j-1}}{\mathrm{d}w_m} + y_{i-j-1} & \text{, wenn } j+1 < i \text{ und } j = m \\ x_{j-i+1} & \text{, wenn } j+2 > i \text{ und } j = m \\ 0 & \text{, wenn } j+2 > i \text{ und } j \neq m \end{array} \right\}$$

So lassen sich iterativ über i alle $\frac{\mathrm{d}y_i}{\mathrm{d}w_m}$ bestimmen.

6.2.2 Bestimmung von $\frac{\mathrm{d}y_i}{\mathrm{d}b}$

$$\frac{\mathrm{d}y_i}{\mathrm{d}b} = \frac{\mathrm{d}b + \sum_{j=0}^{n-1} w_j * a_j}{\mathrm{d}b} = \frac{\mathrm{d}b}{\mathrm{d}b} + \sum_{j=0}^{n-1} \frac{\mathrm{d}w_j * a_j}{\mathrm{d}b} = 1 + \sum_{j=0}^{n-1} \frac{\mathrm{d}w_j * a_j}{\mathrm{d}b}$$

Dabei gilt:

$$\frac{\mathrm{d}w_j * a_j}{\mathrm{d}b} = \left\{ \begin{array}{ll} \frac{\mathrm{d}w_j * y_{i-j-1}}{\mathrm{d}b} & \text{, wenn } j+1 < i \\ \frac{\mathrm{d}w_j * x_{j-i+1}}{\mathrm{d}b} & \text{, wenn } j+2 > i \end{array} \right\} = \left\{ \begin{array}{ll} w_j * \frac{\mathrm{d}y_{i-j-1}}{\mathrm{d}b} & \text{, wenn } j+1 < i \\ 0 & \text{, wenn } j+2 > i \end{array} \right\}$$

6.2.3 Bestimmung von $\frac{\mathrm{d}E_i}{\mathrm{d}y_i}$

Es gilt:

$$\frac{\mathrm{d}E_i}{\mathrm{d}y_i} = \frac{\mathrm{d}\frac{1}{2}(t_i - y_i)^2}{\mathrm{d}y_i} = \frac{\mathrm{d}\frac{1}{2}(t_i - y_i)^2}{\mathrm{d}t_i - y_i} * \frac{\mathrm{d}t_i - y_i}{\mathrm{d}y_i} = (t_i - y_i) * (-1) = y_i - t_i$$

6.2.4 Bestimmung von $\frac{\mathrm{d}E_i}{\mathrm{d}w_m}$ und $\frac{\mathrm{d}E_i}{\mathrm{d}b}$ aus $\frac{\mathrm{d}y_i}{\mathrm{d}w_m}$ und $\frac{\mathrm{d}y_i}{\mathrm{d}b}$

$$\frac{\mathrm{d}E_i}{\mathrm{d}w_m} = \frac{\mathrm{d}E_i}{\mathrm{d}y_i} * \frac{\mathrm{d}y_i}{\mathrm{d}w_m} \text{ und } \frac{\mathrm{d}E_i}{\mathrm{d}b} = \frac{\mathrm{d}E_i}{\mathrm{d}y_i} * \frac{\mathrm{d}y_i}{\mathrm{d}b}$$

6.2.5 Bestimmung von $\frac{\mathrm{d}E_{ges}}{\mathrm{d}w_m}$ und $\frac{\mathrm{d}E_{ges}}{\mathrm{d}b}$

$$\frac{\mathrm{d}E_{ges}}{\mathrm{d}w_m} = \frac{\mathrm{d}\sum_{i=1}^{k} g_i * E_i}{\mathrm{d}w_m} = \sum_{i=1}^{k} g_i * \frac{\mathrm{d}E_i}{\mathrm{d}w_m} \text{ und } \frac{\mathrm{d}E_{ges}}{\mathrm{d}b} = \frac{\mathrm{d}\sum_{i=1}^{k} g_i * E_i}{\mathrm{d}b} = \sum_{i=1}^{k} g_i * \frac{\mathrm{d}E_i}{\mathrm{d}b}$$

6.3 Anlernen des Netzwerks

Zum Anlernen des Netzwerks wurden pro Iterationsschritt zu hundert Funktionen des zu testenden Typs mit zufällig gewählten Parametern die letzten zehn Funktionswerte vor einer zufällig gewählten Stelle in einem zufällig gewählten Abstand als Beispieleingaben und die nächsten fünf Funktionswerte als Beispielausgaben genutzt, um den Fehlergradienten zu berechnen.

6.4 Auswertung der Ergebnisse

Bei quadratischen Funktionen und Sinusfunktionen funktioniert das Anlernen einwandfrei, benötigt aber eine beachtliche Rechenzeit. Interessant ist der Verlauf von Fehlerkurve 1 : es werden recht schnell Werte für die Gewichte gefunden, bei denen der Fehlerwert schon deutlich geringer liegt als bei ihren Ausgangswerten. Dann wird die Veränderung aller Werte viel langsamer. Im weiteren Verlauf wechseln manche Gewichte wie $w9$ sogar noch ihr Vorzeichen, bis sie bei circa einem Drittel der Iterationen schon mehr oder weniger ihre Asymptote erreicht haben. Weil der Fehlerwert aber logarithmiert wird, sinkt der Graph sichtbar weiter bis hin zu einer Sättigung nach circa der Hälfte der Iterationen. Ab dort kann das Netzwerk sich nicht weiter verbessern, weil Zufallswerte zu den Testdaten addiert werden, um Messfehler zu simulieren.

7 Fazit und Ausblick

Beim Umsetzen des eigenen neuronalen Netzwerks wurden einige kritische Punkte deutlich:

- Neuronale Netzwerke erfordern erheblich mehr Rechenleistung als übliche Methoden.
- Die Lernkonstante ist ein kritischer Parameter, der angemessen angepasst werden muss.
- Es sind große Mengen an Lerndaten notwendig.
- Bei der aktuellen Implementierung ist nur die Voraussage einzelner Werte statt einer diskreten Funktion möglich.

Dafür sind neuronale Netzwerke herkömmlichen Methoden in anderen Punkten deutlich überlegen:

- Sie sind ein generellerer Ansatz. Es muss kein Mensch dem Netzwerk angeben, mit einem Polynom welchen Grades eine Ausgleichskurve bestimmt werden soll oder was für eine Art von Funktion verwendet werden soll.
- Es können insbesondere mit tiefen Netzwerken deutlich komplexere Zusammenhänge erkannt werden.
- Neuronale Netzwerke sind problemlos erweiterbar. Es wäre beispielsweise denkbar, dass mehrere logisch zusammenhängende Parameter wie Aktienkurse oder die Sensordaten eines Flugzeugs vorausgesagt werden sollen. Das neuronale Netzwerk könnte dann einfach die letzten Werte aller Parameter erhalten und als Ausgaben die prognostizierten Werte aller Parameter haben. Das würde an der Logik des Netzwerkes wenig ändern, herkömmliche Methoden wären hingegen nicht mehr zu gebrauchen.

Das erklärt, warum neuronale Netzwerke in vielen Anwendungen deutlich an Popularität gewinnen. Rechenleistung und Lerndaten sind immer mehr vorhanden. Auch das Problem der einzelnen Punkte ist durch die Anwendung einer Ausgleichsfunktion auf die extrapolierten Punkte lösbar. In Zukunft werden sie deshalb immer mehr verwendet werden.

8 Anhang

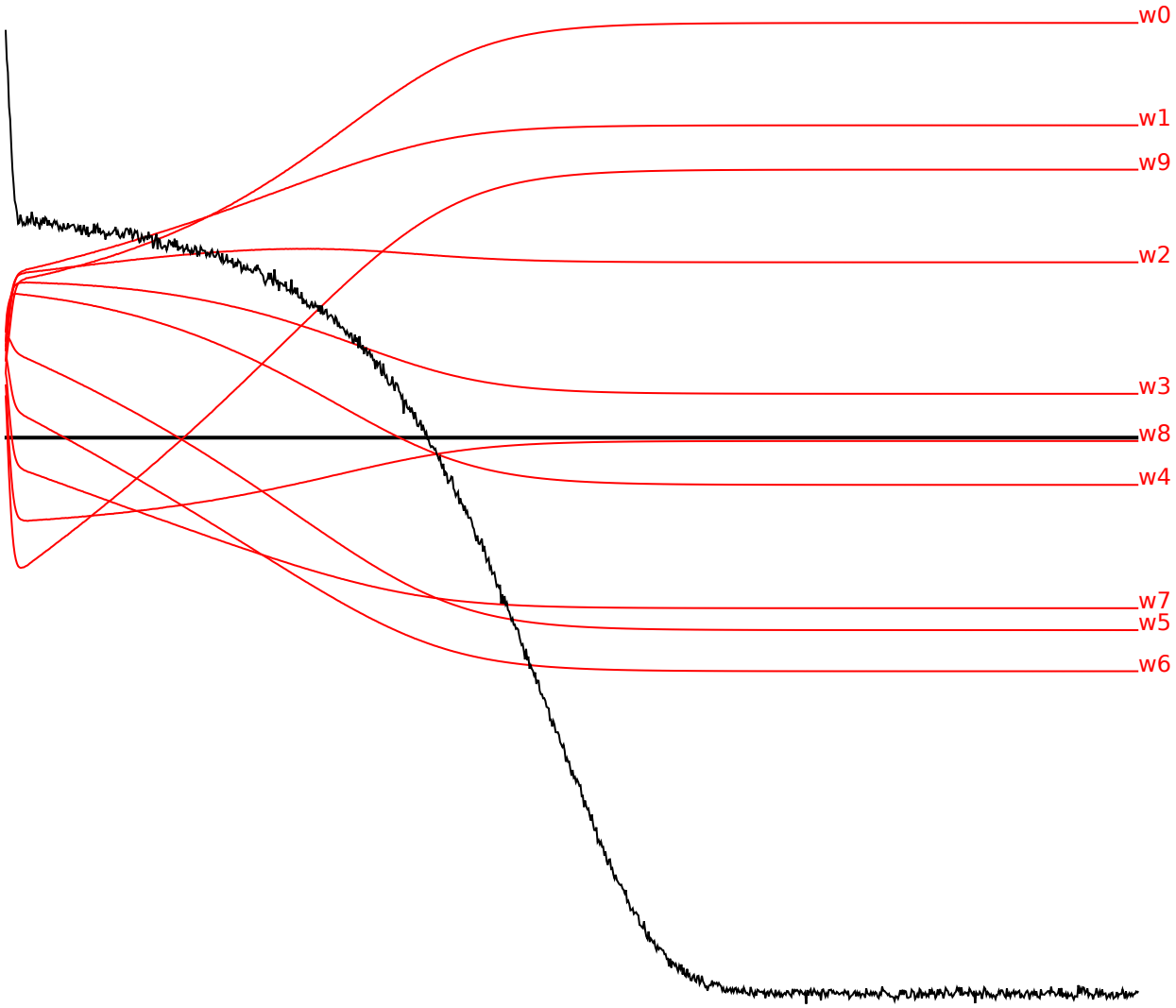

Abbildung 1: Logarithmierter Fehlerwert (schwarze Kurve) und Werte der Gewichte (rote Kurven) über die Anzahl der Iterationen aufgetragen beim Anlernen einer quadratischen Funktion bei einem Netzwerk mit $n = 10$ und $k = 5$

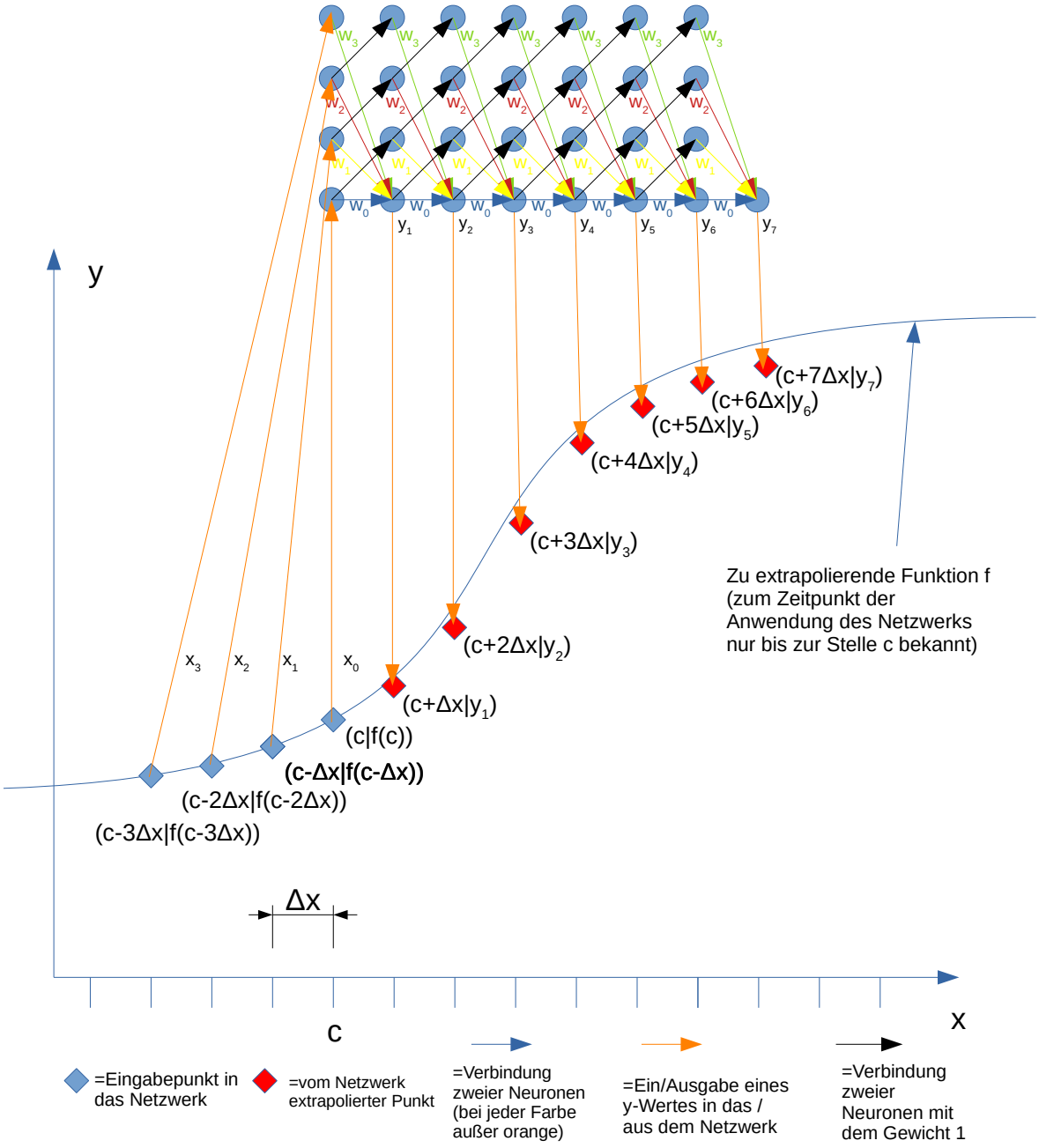

Abbildung 2: Aufbau des eigenen Netzwerkes mit $n = 4$ und $k = 7$. Die Neuronen (blaue Kreise) haben als Aktivierungsfunktion die Identitätsfunktion.

8.1 Quellen

- `https://en.wikipedia.org/wiki/Binomial_coefficient#Partial_sums`
- `http://neuralnetworksanddeeplearning.com/chap4.html`

- M. Mitchell, Tom: Machine Learning, McGraw-Hill Education; 1 edition (March 1, 1997), Kapitel „artificial neural networks“

8.2 Definition der nicht angegebenen Grenzen von Summen

Wenn nur die Laufvariable, aber nicht die Grenzen einer Summe in dieser Arbeit angegeben ist, so ist damit die Summe über alle Werte der Laufvariable gemeint, die definiert sind.

$\sum_n E_n$ ist beispielsweise die Summe aller vorhandenen E_n, die Anzahl der Summanden ist also gleich der Anzahl der Trainingsbeispiele.

Erklärung über die selbständige Anfertigung der Arbeit

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, alle aus anderen Werken wörtlich oder sinngemäß entnommenen Stellen und Abbildungen unter Angabe der Quelle als Entlehnung kenntlich gemacht und keine anderen Hilfsmittel als die angegebenen verwendet habe.

Mainz, den 16. Mai 2019

Ole Petersen